Appendix A
Papers

A.1. Ø. Hanssen, F. Eliassen, *"A Framework for Policy Bindings"*, Proc. DOA'99, Edinburgh, IEEE press, 1999, [Hanssen99].

A.2. Ø. Hanssen, F.Eliassen, *"QoS aware Binding for Distributed Multimedia Systems"*, OOPSLA'99 Doctoral Symposium, Denver, November 1999.

A.3 Ø. Hanssen, F. Eliassen, *"Policy Trading"*, Proc. Distributed Objects and Applications '00, Antwerp, IEEE press, September 2000, [Hanssen00].

A.4 Ø. Hansseen, "Towards Declarative Characterisation and Negotiation of Bindings, in *Proc. Adaptive and Reflecive Middleware 2005*. ACM, 2005 [Hanssen05a]

Towards a QoS aware Binding Model

Ø. Hanssen, F. Eliassen

University of Tromsø, Dept of Computer Science
9037 Tromsø, Norway,
{oivindhl frank}@cs.uit.no

ABSTRACT

In this paper, we present the design of QoSMail, a QoS email application demonstrating a flexible model for using simple declarative requirement-statements and awareness of system properties to dynamically select policies governing message delivery. We believe that these ideas may apply to other cases as well such as binding of stream interfaces. Foundations of the QoSmail design are a framework for describing Quality of Service requirements, a framework for describing the relevant attributes of the environment, i.e. the resources available for transport, and a framework for policy selection, based on QoS-requirements and environment attributes.

Keywords: Binding, policies, Quality of Service, user profile, service profile, email.

1. INTRODUCTION

A problem in open distributed multimedia systems is the creation of bindings between different components, that meet the requirements for Quality of Service from the user or application programmer. A challenge is to achieve adaptability to both user-requirements and changing running-environments. We may imagine tools for mapping from a declarative QoS specification plus some descriptions of the running-environment, to a policy telling how the binding should be set up. A policy would typically denote the choice of protocols, resource allocation strategies etc.

In this paper, we present the design of QoSMail, a QoS aware email application demonstrating a flexible model for using simple declarative requirement statements and awareness of system properties to dynamically select policies governing message delivery. We believe that these ideas may apply to other application areas as well, such as the binding of continuous media streams. Foundations of the design are the following:

- A framework for describing Quality of Service requirements.
- A framework for describing the relevant attributes of the environment, i.e. the resources available for transport
- A framework for policy selection, based on QoS-requirements and environment attributes. In our example this means matching the delivery requirements with an appropriate transport channel, together with a policy for how channels and protocols are used to deliver messages as required.

The rest of the paper is organised as follows. Section 2 compares our approach with some related research. Section 3 introduces the core concepts of flexible binding, policies, the user-profile model to capture application requirements, and a service-profile model to capture system properties. We also show how these concepts are related to each other and how our model can support adaptation to dynamic change in system properties.

Section 4 is a study of a QoS aware email application, demonstrating our concepts through a practical use of user-profiles, service profiles and how those properties can be mapped to a proper policy. A selected policy is implemented during run time by dynamically looking up and downloading corresponding pieces of code representing the policy. This code can carry out delivery of mail with the requested QoS, by using the available resources.

2. RELATED WORK

With few exceptions, research on QoS and adaptability has traditionally focused on system level issues such as networks, and operating systems rather than application level QoS. A notable exception is the QuO architecture (QoS for CORBA objects) developed at BBN/Rome Lab. The goal of this architecture is to support the adaptability to different QoS requirements, changing usage patterns and underlying resources[6]. It extends the functional interface definition language (IDL) with a QoS description language (QDL). This capture application's expected usage patterns and QoS requirements for client's connections to objects.

Both our approach and the QuO approach adopt the Open Implementation Approach [4] to allow object designers to expose key design decisions that affect Quality of Service. This makes it possible to alter the non-functional behaviour of applications by choosing the implementation which is best suited for the situation. Our profile abstractions are inspired by the QoS region abstraction of the QuO project. Two types of regions are introduced:

- Negotiated regions which reflect the expectations of QoS and usage patterns.
- Reality regions which reflect the actual client usage/QoS measured. For one negotiated region there may be many reality regions.

QuO has a layered model of communication. Each layer tries to mask changing regions of the layer below (regarded as reality regions) to a negotiated region presented to the layer above, by using different masking objects. The use of masking objects could be described by using our model of policies.

The QoS broker architecture[5] facilitates negotiation of QoS between application and system, mapping of QoS parameters and orchestration of resource management at different levels (network, hosts OS etc.). Our model of policies capture the resource management which may include all aspects of a communication. The process of selecting and installing policies can be related to the QoS brokerage model, but many of the mechanims of the QoS broker entity will be policy-specific in our approach and indeed.

In the Lancaster University Sumo project[9] QoS is spefified as annotations on exported interfaces. An annotation consists of two parts: (1) The QoS offered and (2) the requirements to the environments. Annotations are formally described in a quality logic (QL), which captures time relations. This allows precise reasoning and mapping to resource management and reactive QoS monitor implementations that control bindings so requirements are met, if possible.

3. CORE CONCEPTS OF FLEXIBLE BINDING

3.1. Bindings and policies

A binding is the association of a client program and the activation of a communication path to a remote object. The binding will be associated to a policy that tells how the binding is activated to meet certain non-functional requirements. The policy typically captures implementation issues like choice of protocols, resource management strategies etc. There are mainly two concerns that lead to the choice of one particular policy when activating a binding:

- The application level requirements for non-functional properties (QoS).
- The type of running environment for the policy implementation expressing available computational resources. A policy may require a certain type of environment.

3.1.1. Meta-policies

There is also a need for policies for how activations of bindings are managed and for the choice of and (possibly) dynamic replacement of policies as a response to changing environment, non-functional requirements and usage patterns. We therefore introduce the concept of meta-policies (policies for using policies). Examples of meta-policies include:

- Passivation of the binding after a certain time of inactivity.
- Changing policies dynamically to adapt to changing resource availability, usage patterns and Quality of Service from the network.

3.2. QoS and User Profiles

The term Quality of Service (QoS) generally captures non-functional properties such as security, dependability, synchronisation, presentation, performance etc. Research on QoS, however, have traditionally focused on properties related to real-time and multimedia communications like performance and synchronisation. Research has also been mostly focused on networks, network protocols, or operating systems[1, 3, 5].

In contrast, we are interested in QoS at a higher level, i.e. the parameters the user or the application programmer deal with. However, to accommodate a QoS-requirement, applications or middleware must rely on QoS delivered from network and operating system services. Thus, high level QoS provision requires mapping from application level QoS requirements to lower level QoS requirements.

Generally the number of possible parameters and metrics to describe QoS requirements are many, and an important issue is how to make abstractions that hide the complexity and irrelevant details. We want QoS requirements to be simple and declarative.

Our approach is to introduce user profiles. A User Profile is an entity which denote a group of requirements for non-functional properties (QoS) which typically appear together. A profile is named by applications to identify a related set of requirements. For instance we may define a profile called *CD-Quality-Audio* which has a qualitative meaning to the user, but which implicitly impose a set of (quantitative) QoS-requirements to the digital audio stream.

We also define a sub-profile relationship. A policy that meets the requirements of a profile X, also meets the requirements of profile Y if X is a sub-profile of Y. For example we would define a profile *CD-Quality-Audio* as a sub-profile of *Phone-Quality* if a digital audio stream satisfying *CD-Quality-Audio*, also satisfies *Phone-Quality-Audio*.

3.3. Service profiles

A service profile is an entity denoting a group of properties of the computing and networking environment (resources available, type of services available, quality of service) which often appear together. We also define a sub-profile relationship for service-profiles. A sub-profile inherits the properties of a super-profile.

Associated to a service profile is a (possibly empty) set of binding policies which require the profile. A policy which requires a service profile *X* could be used with profile *X* or the sub-profiles of *X*.

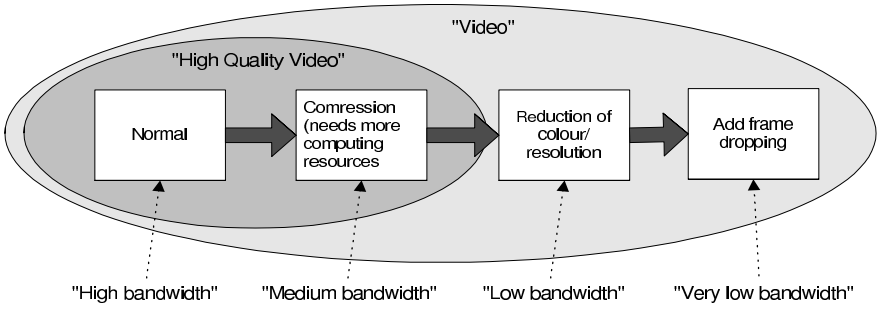
3.4. Dynamic change of policy

Meta-policies specify how bindings may be reactivated with new policies in response to changing system properties, i.e. paths of change.

Change of policy could be initiated from the system due to change of service-profile. This could be done transparently to the client program, as long as the new policy still conforms to the user profile in use. If not, the system needs to renegotiate the user-profile with the client program.

The figure below shows an example of a path of change (degradation path) caused by a reduction of available network bandwidth (service profile is changed from "high bandwidth" to lower bandwidth profiles). This path leads to a change of user-profile from "high quality video" to "video". The "high quality video" is a subprofile of "video". When the video is not high quality, colours and resolution may be reduced and frames may be dropped, by use of filters[7].

Figure 3.1: Example of degradation path

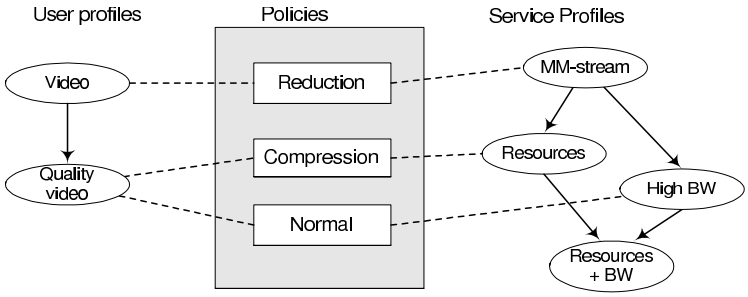


3.5. Profiles and policies - an example

In the following example we have a user profile *Video* with sub-profile *Quality-Video,* and a service profile *MM-stream* which has with two sub-profiles. The profile *Resources* provides computing resources (e.g. for compression) and the profile *high-BW* provides high bandwidth networking. We also include a service profile that provides both computing resources and high bandwidth.

Three policies are available. The *Normal* policy supports *Quality-Video* and requires the *high-BW* service profile. The *Compression* policy supports *Quality-Video* and requires the *Resources* service profile. The *Reduction* policy supports *Video* (reduced quality) and requires the *MM-stream* service profile.

Figure 3.2: Relationship between profiles and policies

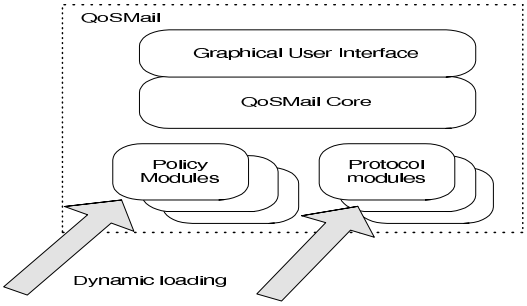


4. THE QOSMAIL APPLICATION

In our case study, the concepts introduced above has been applied to the delivery of email where users have different requirements of for example the secrecy or urgency of messages. The QoSMail application demonstrates the usage of simple declarative requirement-statements and awareness of system properties to dynamically select policies governing message delivery. In our example this means matching the delivery requirements with an appropriate transport channel, together with a policy for how channels and protocols are used to deliver messages as required.

The QoSMail application is designed as a ‘‘microkernel’ that can be dynamically extended, by ‘‘plugging in’ software modules that implement protocols and policies (see the figure below).

Figure 4.1: QoSMail architecure



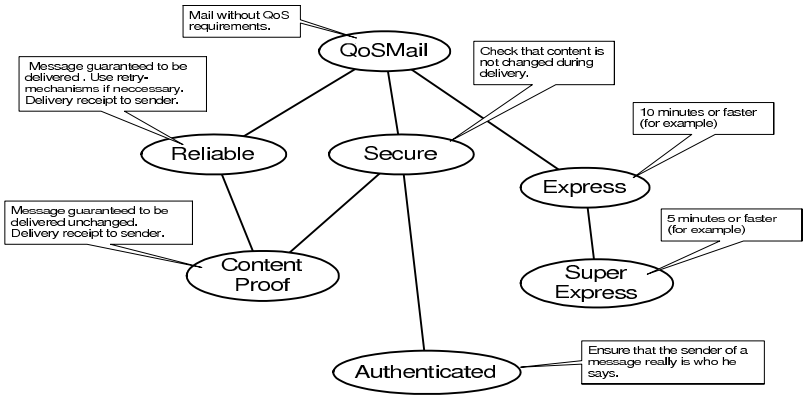
4.1. User Profiles for QoSMail

Many QoS parameters can be associated to email messages. QoSmail recognises the following types of parameters:

- Urgency of message delivery specified for example as a urgency level
- Reliability of message delivery, realised for example as the optional use of delivery receipts returned to the sender.
- Security level of message delivery. The selected level may, for example, trigger the use of encryption techniques or restrict what types of transfer-media can be used.

Figure 4.2 shows an example of user-profiles for QoSMail. Note the *Content-Proof* profile which illustrates how QoS may be expressed as combinations of profiles.

Figure 4.2. Example User Profile Graph



In general different nodes will have different profile-hierarchies. To ensure interoperability, there is a need to standardise (i.e. agree among the users communicating), what the profiles are and what they mean. One might define a ‘‘core’-set of profiles which all nodes know, and allow single users or sub-groups of users add new subprofiles of the core-set (typically combinations). Hence it must be possible to add profiles dynamically at a node.

4.2. Transport resources

Users send messages to other users at specific locations. In QoSMail we use destination-nodes or destinations to identify the recipient’s location. Destinations are grouped into domains. A domain is reached by a channel. Furthermore, domains may have subdomains. A channel that reaches destinations in a domain also reaches destinations in its subdomains.

4.2.1. Channels

A channel is an entity representing the path to a specific domain together with associated resources for transport. A channel may have certain attributes, e.g. media, Quality of Service, supported protocols, other available resources etc. More than one channel may lead to a domain, each of which may have different attributes. When sending a message to a destination, one of the possible channels must be selected, i.e. the one that makes the best fit to the requested quality of service. This suggest a mapping from user-profile, to channel-attributes. This may be done through policies, i.e. the policies specify a user profile and requirements of the attributes of the channel.

4.2.2. Service Profiles

From the set of attributes denoting capabilities of a channel, we can tell if a given policy may use the channel. The attributes of a channel may be expressed as a service profile (c.f. section 3.3). A policy requires a specific service-profile, i.e. it can only use channels which provides that profile or a subprofile.

The profile may change dynamically for a given channel, but only along certain paths in the profile graph. Dynamic change may for instance be initiated by monitors that measure certain QoS parameters. One example is the measurement of delivery time. An example service profile graph for QoSMail is shown below. The thicker edges shows allowed paths of dynamic change.

Figure 4.4: Example Service Profile graph

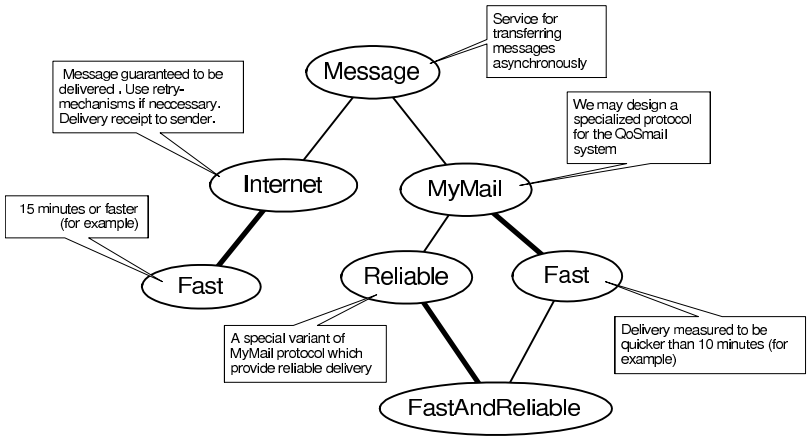
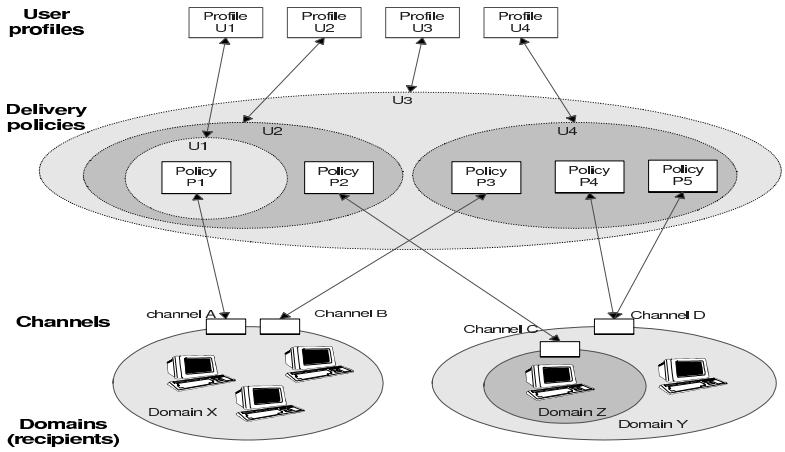


Figure 3.5 illustrates the relationship between user-profiles, policies, channels, destinations and domains. First, a policy conforms to a user-profile. For instance the policy *P1* conforms to the profile *U1*, but also to *U2* and *U3* since *U1* is a subprofile of *U2* and *U2* is a subprofile of *U3*. Each policy can make use of a set of channels. *P1* can only use *channel-A*, which means that if user-profile *U1* is requested, only *channel-A* may be used, and *domain-Y* and *Z* cannot be reached. If *U2* is requested, we can reach *domain-X* through *channel-A* with *P1* and *domain-Z* through *channel-C* with *P2*. If *U4* is requested, all domains can be reached through *channel-B* or *D*.

Figure 4.5: Relationship between user profiles, policies, channels and destinations



4.3. Matching messages with channels and policies

A policy denotes how protocols and other resources should be used to deliver a given message on a given channel, such that the quality of service requirement is met. Policies are represented as software modules which are dynamically loaded into clients when needed. Policies may contain references to protocol-implementations or other resources, which may be dynamically downloaded as well. Different policies may share implementations or parts of implementation. If two policies uses the same implementation the difference may for instance be reflected in parameters used when activating the policies.

Each policy is associated with a user-profile (tells what quality it provides) and a service-profile (tells what it requires from the channel). When a client wants to send a message, it uses the user-profile and the channel's service profile to look up a policy which match those in a policy-repository. The necessary code (and possibly parameters) are then downloaded from there. This code will be used to carry out the delivery.

4.3.1. The selection mechanism

A mechanism for proper selection of channel and policy is the core of our framework for message delivery. We believe that the models of quality of service and channel-attributes as profiles captures just what is necessary to do this selection, provided that the profile definitions are well-defined and agreed on.

Figure 3.6 depicts an example of a configuration of profile-graphs, policies and channels. For example, *domain-X* may be reached by internet mail only (*channel-A*) and domain Y may be reached by internet mail (channel-B) or by using a fast and reliable *MyMail* service (*channel-C*). If a user want to send a message to *domain-X* with user profile *Fast*, *channel-C* and policy *#6* must be used. Policy *#6* provide user-profile *FastAndReliable*, which is compatible with *Fast*. If the destination is in *domain-Y* and user-profile is *Reliable*, policy *#2* may be used (with *channel-B)* or Policies *#4*, *#5* or *#6* may be used with (*channel-C*).

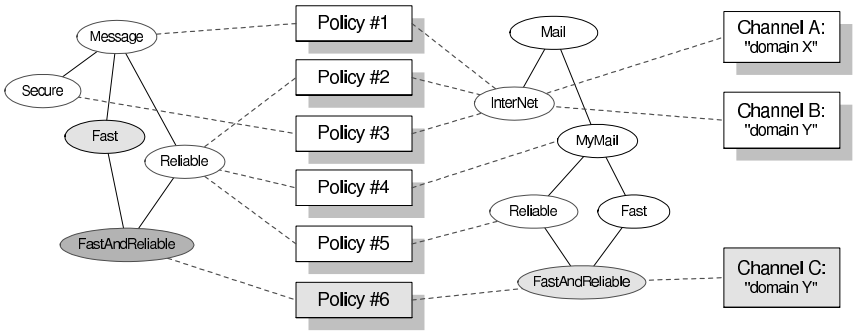
From this example it is clear that more than one policy or more than one channel may be suitable for the delivery of a given message. A simple approach is to select the first match. The algorithm will then be as follows:

1. Search the list of channels and select the first that reaches the destination- domain.
2. Search for the first policy that match the user-profile and the candidate channel's service profile.
3. If such a policy is found, use this (and the channel) to send/receive message, if not, go back to (1) and try the next channel that reaches the destination.
4. If we have tried all possible channels without finding any suitable policy, the search has failed, i.e. the message cannot be delivered.

With this algorithm, the selected channel and policy are not necessarily the best choice. There may be reasons for choosing one candidate before another, for instance cost or efficiency. Therefore, there is a clear potential for improving the

algorithm. A simple (but not perfect) approach may be to order the search of the channels and policies according to some criteria, for instance the cheapest alternative first.

Figure 4.6: Matching of profiles, channels and policies



4.4. Prototype implementation

A prototype QoSmail framework has been implemented. This prototype may be developed further to a working application by adding user-interface facilities, and by implementing usable delivery mechanisms and policies.

The prototype is realised as a Java applet. The applet, the extensions (policies, protocols) and the profile definitions are therefore easily distributed to users by placing them at a web-server and downloading them by using the HTTP protocol.

When a QoSmail applet starts, it loads the profile-definitions plus a list of available policies from a web-server. For each policy in this list we have a name, the full name of the Java-class representing the policy, plus the associated profile names. Channels will typically be defined locally for each user.

To send a message (created and edited by the user) the applet select a channel and a policy (as described in section 4.3). The selected policy refers to a Java class name and when the applet attempts to use this class, the Java run-time system transparently downloads its code from the web-server. The applet uses the class to instantiate a sender-object which takes the message and starts delivering it. The receiving applet will use the policy-class in a similar way to instantiate a receiver-object.

5. CONCLUDING REMARKS

We have introduced a simple QoS-aware model of binding, and we have shown how these ideas can be applied to a particular application domain. The main contributions are as follows:

- An abstraction which captures non-functional application requirements (QoS), the user profile.
- A abstraction which captures the relevant attributes of the environment, typically the transport mechanisms and resources, the service profile.
- A model of policies to capture protocol choices, resource management and other implementation choices, and how we can dynamically select policies for binding or email delivery.

Our model suggest repositories of portable code to represent policies. Since we have simple model of capturing application requirements and environment, it should be relatively straightforward to match these with the appropriate policy and install its implementation. This is closely related to trading[2].

According to Nahrstedt[8] the following services are required in a multimedia environment to provide end-to-end QoS guarantees: (1) Admission and resource allocation for local processor, (2) Admission and resource allocation for network, (3) Negotiation and coordination and (4) Translation between application oriented QoS and resource polices.

If we apply this to our model, allocation is done by policies and admission is done by policy-trading in the sense that a policy is found, only if the requred quality can be achieved, using the available resources denoted by the service profile. Metapolicies may involve negotiation to agree on a policy. Coordination is done by protocols selected by policies, but some coordination may be done by metapolicies in the process of at neccessary places. Translation is done by trading in the sense that trading is mapping from requirements to policy, but much of translation work is actually done statically when defining the profiles and their meaning.

This paper describes a work in progress. In future research we will investigate how well these ideas apply to other application domains and in general. Currently, we study a flexible binding architecture for operational interfaces and we also plan to investigate how the model apply to the binding of continuos media interfaces. Important topics are implementation frameworks, trading architectures and negotiation of between heterogenous endpoints. There is a potential for research in formal models for describing policies and for research in automatic implementation of policies from declarative specifications.

ACKNOWLEDGEMENTS

Much of the work described in this paper was done while the first author was seconded to the ANSA project in Cambridge, UK. This visit was supported by a NATO Science Fellowship through the Norwegian Research Council grant no. 116590/410. We wish to thank to Andrew Herbert, Richard Hayton and Billy Gibson at APM Ltd. for valuable comments related to this work.

REFERENCES

1. C.Aurrecoechea, A.T. Campbell, L. Hauw, ‘‘A Survey of QoS Architectures’, Multimedia Systems Journal, special issue on QoS architecture, 1996.
2. M. Y. Bearman, ‘‘ODP Trader’, Proc. ICODP’93, Berlin 1993, pp. 19-33.
3. Ø. Hanssen, ‘‘FlexiNet - Quality of Service Investigation’, ANSA Phase III Technical report APM.1977.01.00, June 1997.
4. G. Kiczales, ‘‘Beyond the Black Box: Open Implementation’, IEEE Software, 1996, 13(1), p. 8-11 (see also the Open Implementation Home page, Xerox Palo Alto Research Center, http://www.parc.xerox.com/oi).
5. K. Nahrstedt, J. Smith, ‘‘The QoS Broker’, IEEE Multimedia, spring 1995.
6. J. A. Zinky, D. E. Bakken, R. E. Schantz, ‘‘Architectural support for Quality of Service for CORBA Objects’, Theory and Practise of Object Systems (Special Issue on the OMG and CORBA), January 1997.
7. N. Yeadon, A. Mauthe, F. Garcia, D. Hutchison, ‘‘QoS Filters: Addressing the Heterogeneity Gap’, proc. Interactive Multimedia Systems and Services (IDMS’96), Berlin, March 1996.
8. K. Nahrstedt, J. Smith, "A Service Kernel for Multimedia Endstations", in Multimedia: Advanced Teleservices and High-Speed Communication Architectures, lecture Notes in Computer Science No. 869, R. Steinmetz (ed.), Springer Verlag, 1994, pp. 8-22.
9. G. Blair, J.B. Stefani, Open Distributed Processing and Multimeda, Addison Wesley, 1997.

A Framework for Policy Bindings

Øyvind Hanssen *, Frank Eliassen **

* Agder College, Faculty of Engineering, Grimstad, Norway
** University of Oslo, Department of Informatics, Oslo, Norway
Oyvind.Hanssen@hia.no, frank@ifi.uio.no

Abstract:

In this paper we investigate the design of extensible middleware that support dynamic binding configuring by pluggable and replaceable policies. An important aspect of our approach is the distinction between bindings and their activations, which allows us to reason about and implement bindings with changing activations and activation policies (adaptation), and policies for managing activations (metapolicies) as separate entities. A design of a prototype binding framework which supports pluggable policies and metapolicies is described in detail.

1. Introduction

There is growing interest in distributed computing middleware that can *adapt* to different (and changing) non-functional application requirements as well as the service level and QoS from the environments applications are running in. Components of distributed applications are often hard to reuse in different environments and code itself hard to maintain because code that tailors them to specific environments and requirements is not clearly separated from application code.

Our hypothesis is that there should be a separation of concerns between *functional* properties and *non-functional* properties of component interfaces. Such separation of concerns has been proposed by several authors, e.g. [1, 2].

Further, we suggest to relate non-functional properties to *bindings* between components. If components could be implemented independently of non-functional properties, it would enhance reusability of software components and make them easier to implement and maintain. For instance, we might think of binding to a video source interface differently (e.g. colors/black and white or high/low resolution) depending on application requirements and run-time environment (resource availability, network QoS etc). Figure 1 illustrates this principle: The non-functional properties of a binding are provided by a *policy* (for configuring the binding with respect to choice of protocols, transparency mechanisms and resource management). There may also be policies for how policies are selected and installed, or how the binding should adapt by replacing the policy.

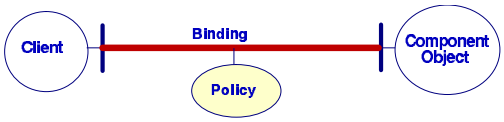


Figure 1. Policy binding

To address the question of how and to what extent this vision can be achieved, we aim to investigate the following problem areas:

- The design of middleware to support dynamic binding configuration by "plugging in" or replacing policies. To achieve this, middleware need to be open, extensible and configurable. This leads to the question how we can open up certain aspects of middleware engineering for policy programmers.
- Foundations for stating requirements and environment properties declaratively, for automated mapping of these to suitable policies, and for specification and/or implementation of policies.

This paper focus mostly at the first problem area. Our approach is to introduce a distinction between *bindings* and their *activations*. Basically, the activation is the configuration of the protocol stack (and associated resources) which manipulate the invocation on its way to the target. A policy determines how an activation should look like, and is represented by a pluggable *activator* component. Bindings can be associated with *metapolicies* (policies for how activations are managed) and represented by a *binding proxy* at each side of the binding. Metapolicies might for instance involve late binding, adaptation or selection of binding policies, either directly, by policy trading or negotiation. Bindings are set up by pluggable *binder* components. We also introduce a way to open up middleware engineering through *PPI*'s (policy programmer interfaces). A proper type hierarchy for PPI's supports pluggability of policies.

The second problem area is addressed by a model for stating QoS requirements and environmental properties. This model is not treated in depth here (see [3]), but in essence it is like a type/subtype model, i.e. a user requres a specific type of QoS and the policy requires a specific type of service from the environment. A subtyping relationship

ensures that a requirement can be met by a compatible offer. Metapolicies which involves policy-trading or negotiation can map from such declarative QoS descriptions to suitable policies.

The paper is structured as follows: *Section 2* introduces a conceptual model of flexible bindings, supporting pluggable and adaptable policies. It defines the main principles of bindings, policies, binding-management through metapolicies and how middleware can expose engineering viewpoint to programmers through PPIs. *Section 3* presents the *FlexiNet* ORB framework developed by the ANSA project. *FlexiNet* is used to realise our prototype binding framework (The *FlexiBind* framework). The design of *FlexiBind* framework is presented in *section 4* and it is an extension to *FlexiNet* which supports dynamic configuration and management of bindings and activations.

2. Foundations

In this section we describe a conceptual model of flexible bindings supporting pluggable and adaptable policies. We introduce the distinction between bindings and their activations, policies and policy management (metapolicies), a model of describing the capabilities and requirements of policies plus some principles for opening up engineering aspects of middleware to policy implementers.

2.1. Policy-governed bindings

A binding can either be *active* or *passive*. When a binding is established, it is not necessarily active. Passive bindings needs to be activated before interactions may be carried out. Activation means that resources are allocated to the object and the communication path between the client and the object. This typically involves loading the object (and its class) into memory, setting up protocol stacks, transparency objects, buffers and other relevant resources. This model allows bindings where the configuration of protocols and resources is not done yet or even where the policy is not known yet. It also allows adaptation of a given binding through re-activation.

Activation is done according to a *policy*, which may be different, due to different ways to implement objects and due to different application requirements for non-functional properties. Examples include:

- Use of a shared standard RPC-channel, e.g. IIOP.
- Reservation or allocation of resources at the end-points and in the network, for each binding, to meet strict real-time requirements of continuous media streams.
- Authentication of the client, the server or both when activating the binding to meet security requirements.
- Use of encryption, using per-activation session keys to meet security requirements.
- Transaction logging to support recoverability.

A *binding* to an object will be represented (at the client side) by a *proxy object* which know how to reach the object's representation if it's active (typically a name/address if it's remote). A passive binding do not have all resources necessary to interact, but it has knowledge of *how* the binding should be activated, i.e. it is connected to a policy.

There is a potential for research on formal models for describing policies, and on methods for automatic implementation of policies from declarative specifications. The idea of *blueprints* in the ANSA FlexiNet project [4] is one approach to this. Our approach is that each policy is represented by a hand written piece of code (for instance a Java class). There is a clear potential for re-use between policy implementations since differences between policies may be small. Careful design of object oriented frameworks for polices, should simplify implementation.

With our distinction between binding and activation, binding establishment do not carry out the policy but may be responsible for selecting and installing a policy to be used for (later) activation. To support adaptation, we may allow the same binding to be reactivated with different policies. Policy management is discussed below.

2.2. Policy selection and management

Metapolicies (policies for policy management) specify how binding policies should be selected, when activation should happen and how bindings may be *reactivated* with new policies in response to changing system properties, i.e. paths of change. Aspects of metapolicies might include:

- Passivation of the binding after a certain time of inactivity.
- Pre-activation of bindings which are likely to be used in near future.
- Changing policies dynamically to adapt to changing resource availability, usage patterns and Quality of Service from the network. This may also include policies for degradation of the QoS delivered to the application.
- Negotiation between clients and servers what policy to use.

In our approach, non-functional (QoS) requirements are specified as *user profiles* while the computing and networking environment are described as *service profiles*. Furthermore, policies are related to user profiles through a satisfaction relationship, and to service profiles through a requirement relationship. Hence a specific policy *satisfies* a set of user profiles (the qualities it provides) and *requires* a service profile (the resources it requires from its environment). Further, there is a compatibility relationship between profiles (subprofiles).

Profiles are simple foundations for policy management. Consider policy selection during binding establishment: It is possible to adopt a *trading* model (c.f. ODP trading [5])

to select pre-existing policy implementations from a repository. When doing *policy trading*, an user-profile and a service profile are given to a trading service which returns a suitable policy, possibly represented as downloadable code fragments. Profiles and policy trading was first investigated in [3].

In adaptation metapolicies, change of policy could be initiated from the system due to change of service-profile. This could be done transparently to the client program, as long as the new policy still conforms to the agreed user profile. If not, the system needs to *renegotiate* the user profile with the client program.

Figure 2 shows an example of a path of change (degradation path) caused by a reduction of available network bandwidth (service profile is changed from "high bandwidth" to lower bandwidth profiles). This path leads to a change of user profile from "high quality video" to "video". The "high quality video" is a subprofile of "video". When the video is not high quality, colours and resolution may be reduced and frames may be dropped.

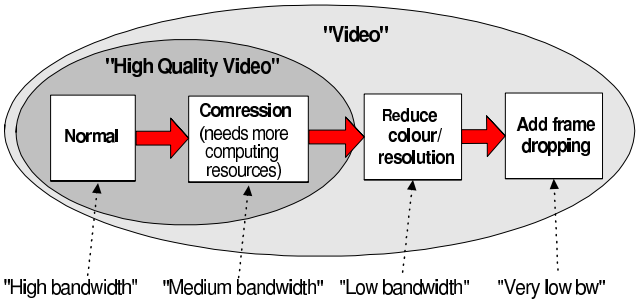


Figure 2. Profiles, policies and paths of change

2.3. Open engineering

The activation of a binding carries out the communication between a client and the target object. It corresponds to the channel concept in the RM-ODP engineering model (including transparency objects). Policies should be able to create and configure activations and therefore it is necessary to "open up" the engineering aspect of middleware so that different policy programmers are able to manipulate the composition of the engineering objects that constitute a channel as well as their resource management policies.

Our approach is to introduce a *PPI* (Policy Programmer Interface) framework. A capsule offer a PPI to policy implementations, separate from the API it offers to applications. PPI's gives access to engineering aspects, but different environments may offer different levels of engineering support and hence they may conform to different PPI-types. A subtyping relationship between PPI-types and standardisation of PPI-type hierarchies are keys to openness and pluggability of policies. A policy requires a particular PPI-type to be able to run, so if the middleware offers a PPI which is a subtype of the required type, the policy can be used with it.

Blair et. al. [6] proposes to apply the principles of open implementation and *reflection* [7] as a principle for opening up engineering aspects of middleware. We follow these principles in the sense that service profiles and PPIs describe and expose aspects of the meta space. They provide access to engineering objects. Thus, the conccept of PPI's are close the concept of *meta object protocols* (MOP). A PPI can be viewed as the meta interface of the *nucleus* while the API represents the base interface.

There is a direct relationship between service profiles and PPI-types since PPI's mirrors an important aspect of what a particular environment is able to perform. Then policy-trading automatically select a policy-implementation which can be used with a given PPI-type.

2.4. Binding management model

Figure 3 summarises the ideas of binding and binding management. The binding is managed by a metapolicy, possibly represented by a controller object. The metapolicy use information from applications and environment and governs selection and installing of policies. Objects representing policies can be viewed as activation factories since they are responsible for setting up activations for the binding. The binding use the activation to carry out invocations. Activations may give feedback to the metapolicy which can be used to decide if adaptation need to take place.

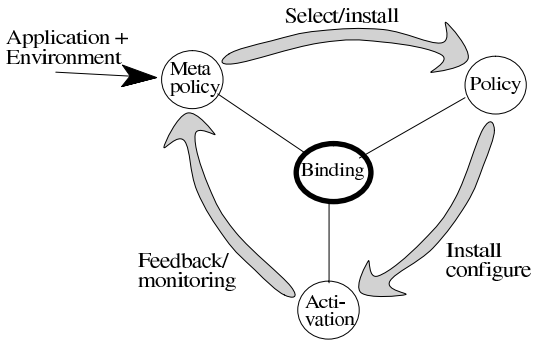


Figure 3. Binding management model

3. The ANSA FlexiNet framework

The *ANSA FlexiNet* platform [4] is a *Java* middleware system built to address some issues of configurable and extensible middleware. It allows programmers to tailor the platform for a particular application domain or deployment scenario. The *FlexiNet* platform can be viewed as a flexible toolkit for creating and (re)configuring ORB's. It provides a generic binding framework plus a set of basic engineering components to populate the framework. By appropriate configuration of components, one can achieve many different middleware facilities, e.g. mobile objects, transactions, security, persistence etc. *FlexiNet* is focused at operational interaction (RPC) but other interaction types (for instance flows) are possible as extensions.

3.1. Basic principles of bindings

An interface on a remote object is represented by a local proxy object, typically a *stub* that converts a typed invocation (method call) into a generic form and passes it through the layers of a protocol stack. Compared with traditional architectures such as *CORBA*, *FlexiNet* puts more of the stub functionality into the protocol stack instead of in the stubs. *FlexiNet* stubs use the *Java* run-time typing information to convert each invocation into a generic form and let the layers in the stack do the rest. This makes stubs so simple that they can easily be generated at run-time by using introspection on the interfaces. However, protocol stacks are more complex and include higher level functions such as serialisation, replication, object management etc.

Protocol stacks and layers. The layers of the *FlexiNet* communication stack can be viewed as reflective objects that manipulate the generic invocation in different ways before it is invoked on the destination object. There is no need for interface specific skeletons, just a generic function that converts a generic invocation object to specific calls on target interfaces. Higher level transparency objects can also be regarded as layers in this architecture.

The generic form of the invocation allows simple interfaces to bind layers together. At the client side, layers implement the CallDown interface which has one operation:

```
public void calldown (Invocation inv)
```

Server side layers implements the CallUp interface:

```
public void callup (Invocation inv)
```

Each layer forwards the generic call to next layer in the stack by calling the calldown or callup method recursively. The invocation object may be manipulated at each layer until it reaches the point where the call either is converted to messages to be transmitted over the network or to a call to the target interface. Returns of the call carry result values with the invocation object in the opposite direction. This model requires that the communication between layers are in form of request-reply pairs. Below the RPC protocol layer, we can only talk about unrelated messages going up or down, so at that level, each layer must implement two interfaces: MessageUp and MessageDown.

An example configuration. The first protocol configuration that was realised in *FlexiNet* was named "green" (by convention, *FlexiNet* protocols are named after colours). and based on the *REX-protocol* [8]. This configuration illustrates what a typical RPC configuration look like and is shown in figure 4 below. Most of the layers shown supports both clients and servers at the same time, so most of a protocol stack instance can be used both as client and as server.

A generic call first goes from a stub to the *client call layer* which acquires a session. The *serial layer* serialise methods, their arguments and results. The *name layer* takes the server interface *identifier* (which is a part of a name of a remote object) and writes it to the output buffer. At the server side, the name layer reads this identifier and use it to look up the target object (name layer represents a mapping from identifiers to target objects). The server call layer use information from the generic invocation object to generate a method call on the correct target object. The *rex layer* (and the session layer) provides RPC semantics over an unreliable message transfer service.

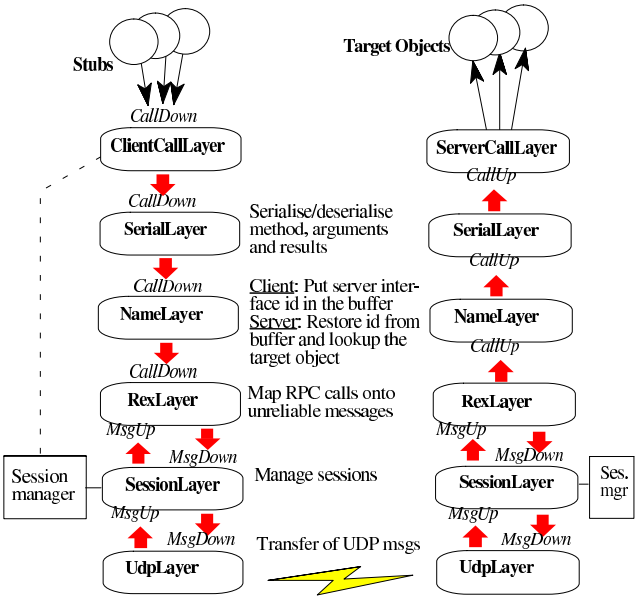


Figure 4. Example protocol in FlexiNet

Sessions. Many RPC protocols maintain state across a number of invocations. For instance, an UDP based protocol (like REX), may need to keep track of unacknowledged replies, and a TCP based protocol may need to maintain a connection. *FlexiNet* provides sessions as an abstraction for managing such information. Sessions are also used to provide concurrency control in the protocol stack, essentially by using per-session locking. A session is typically related to a client thread's association to a server, i.e. at the client side there is typically one session per server (per thread) and at the server side there is typically one session per client (thread).

When messages arrive from the network or invocations arrive from stubs, to a protocol stack, they first need to be associated to a session. In the above example, the *client call layer* acquires a session object from a *session manager*. For downgoing messages, the *session layer* writes a *session identifier* to the output buffer which is read by the receiving session layer and used to look up (or create) the session object there. A session object typically contains the RPC protocol state, but also a dictionary to be used by higher layers to store session related information.

3.2. Protocols and binders

The concepts of names, protocols and binders are keys to extensibility of *FlexiNet*. First the *FlexiNet* architecture allows different types of names. Binding at the server side means generating a name for the interface to be exported (and registering a mapping between the name and the target). A name contains the information needed to allow clients to bind to the target. Names consists of a *protocol name* and protocol specific information needed to locate the target. This is typically a port address plus a identifier for the interface, either a identifier relative to the server port or a globally unique identifier.

To support different protocols, different *binders* can be provided. At the client side binders *resolve* names of remote objects into proxy-objects (stubs), together with the proper communication stacks (which typically should be shared between bindings). At the server side they *generate* a name for a target interface. Thus, there are two types of binders: Resolvers at the client side and generators at the server side.

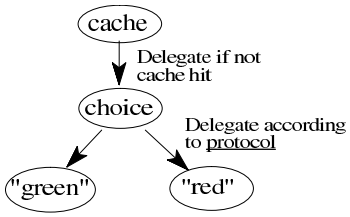


Figure 5. Binder delegation hierarchy

FlexiNet allows many binders and protocols to exist within the same process and there is therefore necessary to select the correct binder to be used for binding. Each protocol stack instance therefore contains references to binders to be used for generating and resolving names or interfaces passed as arguments or results of invocations. Binders can also be arranged into hierarchies, to factor out common functionality or to allow dynamic selection of a binder to be used for a particular binding. Figure 5 shows a simple (and static) example of a binder setup.

More dynamic selection is possible. In addition to the protocol part of the name, applications may also use QoS requirements as arguments to the binder to select the binder which best matches the requirements. Delegating binders can e.g. look up another binder to delegate to, or even negotiate with the server to do the selection.

4. FlexiBind - an experimental framework

We have designed an experimental framework to demonstrate some of the principles described in section 2. Our framework extends the *FlexiNet* architecture. The *FlexiBind* framework can further be extended to support a family of protocols. Protocols and policies can be "plugged" into the middleware dynamically. The framework allows late binding, explicit binding and dynamic adaptation of protocol stacks.

In this section, we go through the most important aspects of the *FlexiBind* framework: The concept of activation (section 4.1), the concept of bindings and binding proxies (section 4.2), the concepts of encapsulating the environment (section 4.3) and the framework for pluggable binders and activators to add protocols, policies and metapolicies (section 4.4). Section 4.5. shows an example of a metapolicy.

4.1. Activations

A principle of the *FlexiBind* framework is the distinction between bindings and their activations. Binding can then be done without knowing (yet) all about how the binding should be activated. This distinction allows *lazy* activation (wait until the first invocation is made), *explicit* activation or even *per-invocation* activation.

The activation object. An activation object represents a composition of layers to be used to carry out interactions. Activation objects are (in principle) constructed at each invocation and is carried with the generic call object through the layers. Each activation object contains an ordered list of references to the layers of the activation. When a call has gone through a layer, the activation object is consulted to get to the next layer.

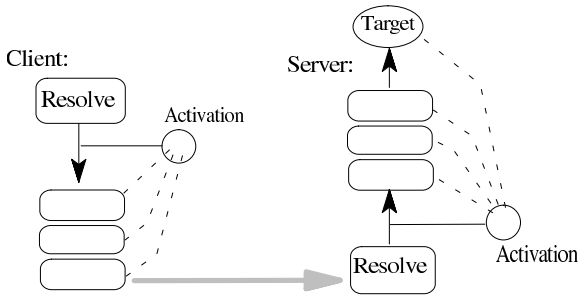


Figure 6. Resolving activation objects

Figure 6 illustrates how the layer composition is determined by resolving to activation objects. At the client side, the target name (on the invocation) is resolved to an activation object which is used to perform the invocation. At the server side, a target name (passed over by the protocol) is resolved to an activation-object that prepares and invokes the operation at the target interface (which is the last part of the resolved activation).

Channels. When a server exports an interface by generating a name for it, some protocol information must be passed along with it such that clients know how to bind to it. In our context, the purpose of the protocol part of a *FlexiNet* name should then be to identify the minimum protocol support needed to activate a binding to the exported interface.

This observation suggest that the activation (protocol stack) should be divided into two parts: A protocol

dependent part which is identified in names generated by servers, and a protocol independent part which can be resolved/activated dynamically. At the server side, the protocol dependent part should normally be the minimum needed to listen for incoming calls and to do dynamic activation. At the client side, this means that we know what the protocol-dependent part of the activation should be at binding time, but it doesn't necessarily have to be activated before the rest of the stack is activated.

It is useful to encapsulate common protocol dependent configurations as *channel* objects. The capsule environment (reflected by a PPI) should offer access to one or more default shared channel objects and/or an interface to instantiate channel objects.

Sharing. Layers (and associated resources) may be shared between bindings. There can be per-binding layers or layers that are shared between all bindings for an interface, object, capsule, cluster or session. Each layer of an activation can be shared differently. In a typical RPC configuration, the channel is capsule-shared (except the RPC protocol state where we need one instance per session) and the upper layers are per binding. How layers could be shared depends on what kind of state they contain (including resources like buffers).

4.2. Bindings

A binding-proxy represent a binding which is not necessarily active. When activated, a binding proxy have an *activator* object attached to it, which is responsible for managing the layers and other resources forming the activation and return them as an activation object to each invocation. The class of the activator represent the policy (c.f. section 2.2.1) and activators may be installed or replaced dynamically. Each binding-proxy implements the interface Binding which defines operations to explicitly activate, passivate or to set or replace the policy (represented by activator classes). Figure 7 shows the class design for binding proxies:

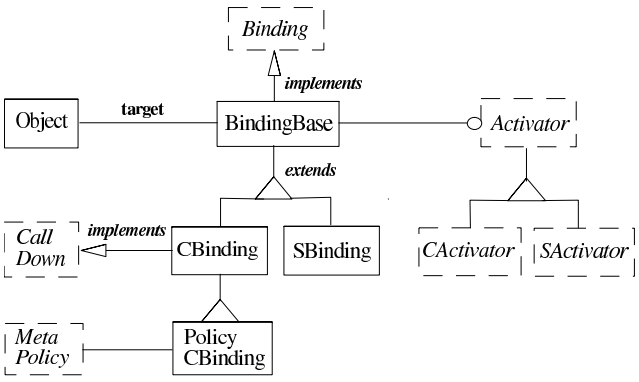


Figure 7. Class diagram for bindings

Client side bindings. Client side binding objects (CBinding) implements the generic invocation interface (CallDown) and can therefore be regarded as a layer. When a binding is established, it is represented by a stub which contains a CBinding, where the target is a name (Address).

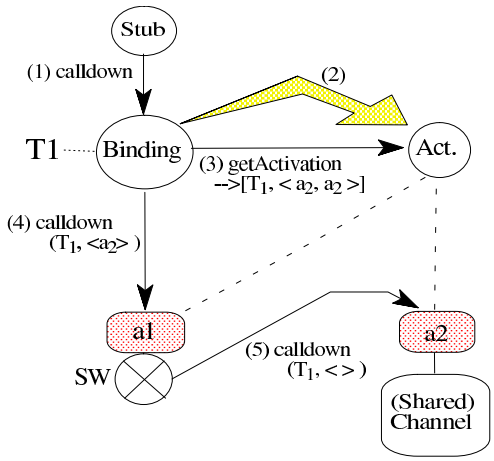


Figure 8. Client side binding - Activation, resolving and call scenario

Figure 8 illustrates how the binding works (se also section 4.2.3). When receiving a generic invocation from the stub (1), the CBinding asks the activator by calling its getActivation method, for an activation object (3) which contains references to the layers to which the generic invocation is forwarded (4). If necessary (typically at the first invocation) the binding need to activate (2) by instantiating the activator (this can also be done explicitly via the Binding interface).

Each of the layers in the activation's list, except the last one, returns to a switch layer which forwards the invocation to the next layers in the list (5).

Policy bindings. The PolicyCBinding class is a subclass of the CBinding class, which contains a reference to a replaceable metapolicy object. Before and after its own activate and passivate method, a PolicyCBinding object calls the meta-object to policy specific processing. Each meta-object implements the MetaPolicy interface which define the pre_activate, post_activate, pre_passivate and post_passivate methods. Those methods can for instance negotiate with the server to agree on the policy (the class of the activator). Some policies may use one meta-object per binding-proxy and others may share meta-objects between binding-proxies.

Server side bindings. A binding layer (c.f. FlexiNet name layer) is always a part of a server channel. At each incoming call, the binding layer read an interface-identifier from the input buffer (the id is a part of a name) and use it to look up an activation object. To do the lookup, the binding layer consult a binding manager which contain mappings

from interface-id's to binding-objects representing the targets. When exporting an interface, the server create a binding-proxy, an identifier and a mapping between the identifier and the binding-proxy.

Figure 9 illustrates the activation model, by a scenario of how an incoming invocation is treated: When the binding layer asks the binding manager for an activation for a given identifier, the binding manager looks up the binding proxy and asks it for the activation (1). The activation for a binding is managed by an *activator* object if its active. If a binding is not active it needs to activate by instantiating the activator object (2) before it asks it for the activation by calling the getActivation method (3). Typically, the activator sets up the layers of the activation when instantiated and return references to them through the return value of the getActivation method. At the server side the *getActivation* method has the following signature:

```
public ServerActivation getActivation
     (Dictionary session, Object target)
```

The call which now contains a valid activation object then goes to the a *switch layer* (4), which calls the first layer on the activation object's list of layers (5). The call goes back to the switch which calls the next layer on the list. Before a layer is called, the reference to it is removed from the list. When the list is empty, the operation on the target object is invoked (6).

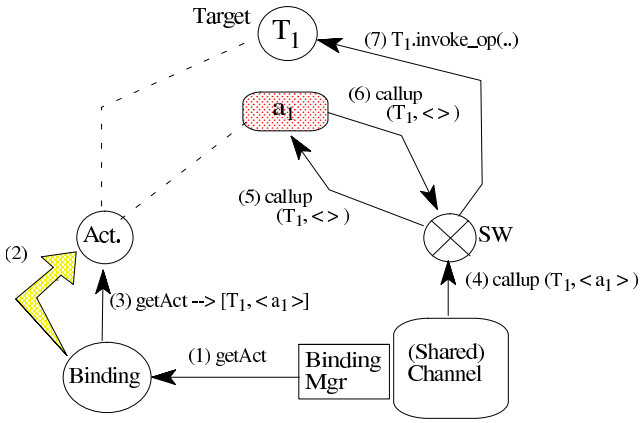


Figure 9. Server side binding - activation, resolving and call scenario

Server side session management. Some servers may need to allow different activations to the same target at the same time and they may need to associate an activation to the session that created it as figure 10 illustrates. The reason for this is that different clients (or client threads) may need to bind to the same server interface in different ways (using different policies). Different bindings to the same target may have different activators which creates different activations. To facilitate this, we register session specific bindings in separate dictionaries instead of in the binding manager. All session-objects in *FlexiNet* contains a dictionary and each invocation (callup) carries a reference to the session object. Session specific bindings are put into the session dictionary and session independent bindings are placed in the binding manager's dictionary. When the binding-layer asks for a binding, the manager first looks in the session dictionary (bindings registered here are visible only by invocations belonging to the same session) and if no binding is found there, the session independent dictionary is used (bindings registered here are visible by all invocations).

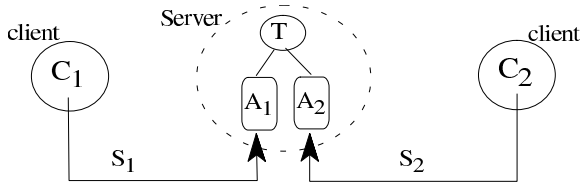


Figure 10. Different activations (A_1, A_2) for different sessions (S_1, S_2)

When binding layer asks bindings (via the binding manager) for activations, the session is always an argument. It is up to the attached policy if the session is used to register session specific bindings.

4.3. Environment framework

To encapsulate *nucleus* services of middleware and to offer *API* and *PPI* (c.f. section 2.3), we introduce *environment objects*. There is one instance per capsule. The middleware is initialised by instantiating an environment object. The instantiation code installs default binders and other resources like shared channels or factories for creating channels.

As figure 11 illustrates, an environment class implements an API and a PPI interface. The *FlexiBind* framework provides Env, a base class for environment classes. *Env* provides method for installing and initialising a default binder. Subclasses of Env add other services/access to engineering objects.

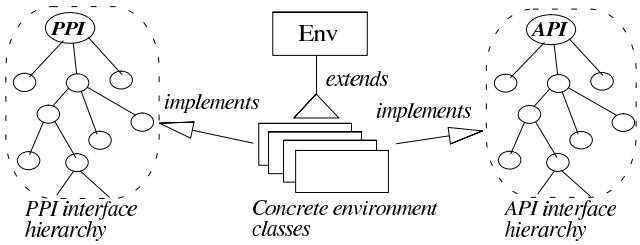


Figure 11. Environment objects and their interfaces

A capsule is initialised by installing an environment. This is done by calling the static method install of the environment class to be used. Its signature is as follows:

```
public static void install
  ( String sbinder, FlexiProps s_args,
    String cbinder, FlexiProps c_args )
```

As arguments, we use the name of the server binder class (sbinder), the arguments (a property dictionary) to be passed to the server binder (sb_args), the name of the client binder class (cbinder) and the arguments to the client binder (cb_args). The install method basically create an instance of the environment class and keeps a static reference to it. The PPI or API interfaces of this object can be reached by calling one of two static access methods:

```
public static PPI getPPI ( )
public static API getAPI ( )
```

4.4. Pluggable binder framework

A *FlexiBind* based ORB can dynamically be configured for different policies and metapolicies by plugging in binder- and activator- components. Those components are implemented as relatively simple extensions of the binder-framework:

Binders. A binder represents the metapolicy and it knows a specific protocol (in the *FlexiNet* sense). In our context this means both how to bind to the server and how to "negotiate" an activation policy between the client and the server. The binder is responsible for instantiating and configuring the binding-proxy with activators or activator selection policies (and possibly connect it to a metapolicy object). At the client side it will be connected to a stub and at the server side it will be registered in the binding manager.

In the *FlexiNet* framework, client binder classes implements the Resolver interface and server binder classes implements the Generator interface. Our framework closely follows this scheme. The framework provide abstract base classes to allow sharing of (common parts of) implementation.

At the client side, a binder object is responsible for resolving names of remote interfaces to binding proxies that can be used to invoke their operations. At the server side, a binder object creates a name for the interface to be exported, and establishes an association between the name and the target interface through a binding-proxy.

A concrete binder class is implemented as a subclass of the binder base class (CBinderBase or SBinderBase). The example below shows how a simple server binder can be written. The constructor extracts the name of the activator class from the argument and converts the type of the PPI interface. The generateName method does the actual binding. It creates and configures a binding-proxy and it generates a name for the interface. A mapping between the identifier part of the name and the proxy is registered at the binding manager. The PPI interface is used to get the identifier generator, the binding manager and the port address of the shared channel.

```
public class Simple_SBinder
   extends SBinderBase
{
   String _act;
   SharedChannelPPI _ppi;
   public Simple_SBinder
      (FlexiProps arg, PPI ppi)
   {
      _act = arg.getProperty("activator");

      if (ppi instanceof SharedChannelPPI)
         _ppi = (SharedChannelPPI) ppi;
      else
         throw new FlexiException
            ("Incompatible PPI");
   }


   public Name generateName
      (Object o, Class cls, FlexiProps qos)
   {
      // Create binding proxy
      SBinding p= new SBinding(_act, o);

      // Get unique identifier
      int id = ppi.getNameGenerator().newId();

      // Register mapping
      ppi.getBindMgr().put(new Integer(id),p);

      // Create and return the name
      TrivName n = new TrivName( PROTOCOL,
         _ppi.getAddr(), id);
      return n;
   }

    ...
}
```

A corresponding resolveName method could be as simple as the example below. Note that the base class performs the stub creation since this is common to all client binders.

```
public CallDown resolveName
   (Name name, FlexiProps qos) throws BadName
{
   return new CBinding(name, _act);
}
```

Concrete binder classes need to provide a constructor method with two parameters: (1) PPI interface. The constructor should test if this is of a the PPI type that the binder required and do a type cast. (2) a property list (FlexiProps) containing parameters, for instance a default activator class name like in the example above.

As the example code above also shows, the name of the activator class is simply passed to the binding proxy constructor. We could use a default value which is hardwired into the code or set by the binder constructor. The resolveName and generateName methods also take a property list (FlexiProps) as argument. This could contain a activator class name or more declarative QoS requirements which is used to look up a suitable activator class. If policy trading/negotiation is used (c.f. section 2.2), the QoS list should contain the service profile and the user profile.

Activators. An activator object is responsible for allocating necessary resources and configuring the activation. Server activators implements the SActivator interface and client activators implements the CActivator interface.

Activator constructors have one parameter: A PPI interface, which is treated the same way as in binder constructors.

The listing below shows a simple example of a server activator. It adds a (stateful and non-shared) layer between the channel and the target which logs information about the invocations going through it. The logger layer is instantiated when the activator is instantiated and a reference to it is returned by the getActivation method. The logger is initialised with a string describing the target and the layer above which is the switch, provided by the environment

```
public class Log_SActivator
   implements SActivator
{
   private SharedChannelPPI _ppi;
   private CallUp _layer;

   public Log_SActivator(PPI ppi, Object t)
   {
      ...

      // Create logger layer
      _layer = new Logger(t.toString(), null,
            _ppi.getSwitch());
   }


   public ServerActivation getActivation
      (Dictionary session, Object t)
   {
      return new ServerActivation(t, _layer);
   }
}
```

4.5. SimpleNeg - an example metapolicy

It is likely that an aspect of many metapolicies will be that the client installs or negotiates a policy on the server during binding. Here, we show the design of a metapolicy where client activates the server (installs a policy). This might be extended to a more sophisticated protocol for negotiating the policy to be used.

Server side. The server binder export the Binding interface together with the target interface so that it can be invoked remotely. This is done by generating names containing two identifiers. One for the target itself (base) and one for the Binding interface (control). These map to two binding proxies (base-binding and control-binding). Client metapolicies can use the control interface to activate the binding for the target interface.

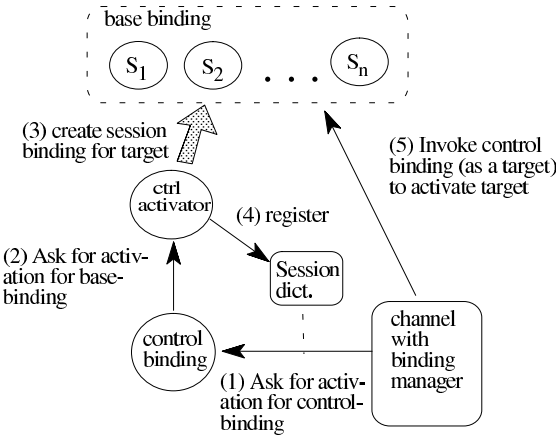


Figure 12. Server side support for remote binding management

To allow different clients to activate their bindings differently at the same time, there must be one base binding-proxy per session. The control-binding's activator creates the session-specific binding-proxy the first time an invocation asks the control-binding for the activation for a given session and register it in the session dictionary (c.f. section 4.2.4). Figure 12 shows what happens when a client invokes the control interface to activate the binding to the target.

Client side. At the client side the binder create instances of the PolicyBinding class. Each binding proxy is connected to a MetaPolicy object (c.f. section 4.2.2) which in the pre_activate method invokes the activate operation on server side control interface (Binding). The post_passivate method calls the passivate operation at server. Both the client side activator class and the server side activator class are given to the binder constructor or the resolveName method (c.f. section 4.4.1) as arguments. Figure 13 illustrates how the client binding initiates activation at both sides.

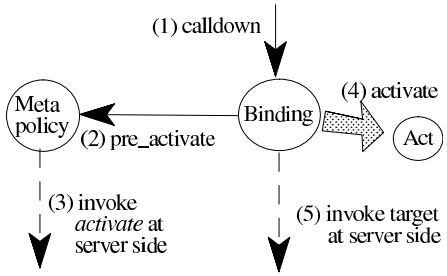


Figure 13. Remote binding management (client)

4.6. Related research

Our work builds on the idea of extensibility via policy bindings, which is not new. For instance the Open OODB project [9] aims to develop an architecture for extensible database middleware. Here, operations like object access

or selection may be associated with invariants (requirements), which can be satisfied by different policies. Policies are realised by policy-performers and managed by policy managers (hides the choice of policy). Besides the database research, we are also influenced by research on extensible operating systems [10, 11], and Quality of Service [12].

Other work on extensible ORB's include the ANSA DIMMA platform [13] which is the result of investigating ORB support for multimedia. DIMMA was designed as a "microkernel" ORB on which one could add personalities (e.g. CORBA) at the top and where one could "plug in" new protocols at the bottom. It was also designed with flow interfaces and resource control in mind. The ANSA *FlexiNet* platform (see section 3) is based on lessons learned in the DIMMA project and both architectures are heavily based on RM-ODP concepts. *Jonathan* [14] is a *Java* implementation of the *ReTina* architecture and is a "microkernel" ORB like DIMMA.

In the *Open Implementation* approach, Kiczales [2] proposes that non-functional issues should be dealt with separately from functional issues and exposed to programs in separate interfaces (Meta Object Protocols). The QuO (Quality objects) architecture [1] adopt this approach to allow object designers to expose key design decisions that affect Quality of Service. This makes it possible to alter the non-functional behaviour of distributed applications by choosing the implementation which is best suited for the situation. QuO extend the functional interface definition language (IDL) with a QoS description language (QDL) which captures application's expected usage patterns, QoS requirements and resource usage, for bindings.

Blair et. al. [6], fully exploits the concept of reflection [7] to provide configurability and openness. Here, the engineering viewpoint is exposed as *meta spaces* which can be associated with every object or interface. Meta-space objects may also provide their own meta-spaces recursively. A framework for components to populate the reflective architecture is being developed. Such components will span from protocol layers to complete bindings or preconfigured environment metaspaces.

5. Conclusions

In this paper we have investigated the design of middleware to support dynamic binding configuration by pluggable and replaceable policies. An important aspect of our approach is the distinction between *bindings* and their *activations*, which allows reasoning about bindings with changing activations and activation policies (adaptation), and policies for managing activations (metapolicies) as separate entities. This leads to two types of dynamically "pluggable" policy components: (1) *Binders* which represents the protocol and metapolicy and (2) *activators* which represents the policy for how protocol stacks and associated resources should be configured.

We have described the design of a prototype binding framework (*FlexiBind*) which is an extension of the *ANSA FlexiNet* framework. Here, bindings are represented by binding-proxies which are attached to an activator and (possibly) a metapolicy object. It also provides open interfaces for policy programmers to middleware engineering through PPI's (policy programmer interfaces). A proper type/subtype hierarchy for PPI's is a key to openness. A policy which requires a PPI type can be plugged into a platform that offers a compatible PPI. An *environment component* is used to initialise the middleware, encapsulate the nucleus components, install initial binders and channels, and provide two interfaces: The PPI for policy programmers and API for application programmers.

To be really useful, the *FlexiBind* framework need to be populated by metapolicies that involves mapping from user requirements and environment parameters to suitable policies. In some cases such information may be gathered dynamically and used by dynamic adaptation policies. We have introduced policy-trading to select suitable policies plus a simple model for user- and service-profiles to reflect requirements and environment. In future research we plan to investigate the use of policy-trading and client-server negotiation with the *FlexiBind* framework, based on our model of profiles and policies.

6. Acknowledgments

The work described here was partly performed when the author was seconded by the University of Tromsø to the ANSA *FlexiNet* project in Cambridge, UK in 1997. This stay was supported by a NATO Science Fellowship through the Norwegian Research Council grant no. 116590/410.

Thanks to Dr. Richard Hayton and Dr. Andrew Herbert at APM (now Citrix), for valuable comments during this stay. Also thanks to Gordon Blair, Geoff Coulson, Fabio Costa, Katia Saikoski and the other researchers from the universities in Lancaster, Oslo and Tromsø who was involved in the CORBAng project.

7. References

[1] J.A. Zinky, D.E. Bakken, R.E. Shantz, "Architectural Support for Quality of Service for CORBA Objects", Theory and Practice of Object Systems (Special issue in the OMG and CORBA), January 1997.

[2] G. Kiczales, "Beyond the Black Box: Open Implementation", IEEE Software, 1996, 13(1), p. 8-11.

[3] Ø. Hanssen, F. Eliassen, "Towards a QoS aware Binding Model", Proc. SYBEN '98, Zurich, May 1998.

[4] R. Hayton et. al. "FlexiNet Architecture Report", ANSA Phase III report, February 1999.

[5] M. Y. Bearman, "ODP Trader", Proc. ICODP'93, Berlin 1993, pp. 19-33.

[6] G. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. Middleware '98, Springer Verlag, 1998.

[7] G. Kiczales, J.D. Riveres, D.G. Bobrov, "The Art of he Metaobject Protocol", MIT-Press, 1991.

[8] D. Otway, E. Oskiewicz, "REX: a remote execution protocol for object-oriented distributed applications", IEEE Press 1987.
[9] D.L. Wells, J.A. Blakeley, C.W. Thompson, "Architecture of an Open Object-Oriented Database Management System", IEEE Computer, October 1992.
[10] C. Small and M. Seltzer, "Structuring the Kernel as a Toolkit of Extensible, Reusable Components", Proc. 4th International Workshop on Object Orientation in Operating Systems, August 1995, IEEE press.
[11] Y. Li, S.M. Tan, M.L. Sefika, R.H. Campbell, W.S. Liao, "Dymamic Customization in the Choices Operating System", Proc. Reflection'96, April, 1996.
[12] C. Aurrecoechea, A.T. Campbell, L. Hauw, "A Survey of QoS architectures", Multimedia Systems Journal, special issue on QoS architecture, 1996.
[13] D. Donaldson, et. al., DIMMA - A Multi-Media ORB, Proc. Middleware '98, Springer Verlag, 1998.
[14] B. Dumant, F. Horn, F. Dang Tran, J.B. Stefani, "Jonathan: an Open Distributed Processing Environment in Java", Proc. Middleware '98, Springer Verlag, 1998

QoS aware Binding for Distributed Multimedia Systems

Øyvind Hanssen, Frank Eliassen

Abstract

We investigate the design of extensible middleware that support dynamic binding configuring by pluggable and replaceable policies. We develop foundations for stating QoS requirements and environmental properties which supports automatic mapping to binding policies.

1. Introduction

There is growing interest in distributed computing middleware that can adapt to different (and changing) non-functional application requirements as well as the service level and QoS from the environments applications are running in. Components of distributed applications are often hard to reuse in different environments and code itself hard to maintain because code that tailors them to specific environments is not clearly separated from application code.

Our hyphothesis is that there should be a clear separation of concerns between functional properties and non-functional properties of component interfaces. Such separation of concerns has been proposed by several authors e.g. Kiczales [1], Zinky et al. [2]. This could also be regarded as an application of aspect oriented programming [3] to middleware, where we identify two aspects of binding to a component; the functional and the non-functional.

Further, we suggest that non-functional properties should be related to bindings between components. Our goal is to develop the foundations for a QoS-aware binding facility that can map from application requirements plus a description of the available system properties to a suitable binding. We are also interested in bindings that can adapt to changing system properties by changing its protocols, resource policies etc. To address the question of how and to what extent this vision can be achieved, we aim to investigate the following two problem areas:

- The design of middleware to support dynamic binding configuration by "plugging in" or replacing policies. To achieve this, middleware need to be open, extensible and configurable. Related work on extensible or reflective middleware include Blair et. al [6] and Hayton et. al. [7].
- Foundations for stating requirements and environment properties declaratively, for automated mapping of these to suitable policies, and for specification and/or implementation of policies.

2. Approach

The first problem area is approached by introducing a distinction between bindings and their activations. Basically, the activation is the configuration of the protocol stack (and associated resources) which manipulate invocations on their way to the target-object. A policy tells how an activation should look like and is represented by a pluggable activator component. Bindings can be associated with metapolicies (policies for how activations are managed) and represented by a binding proxy at each side of the binding. Metapolicies may for instance involve late binding, adaptation or selection of

binding policies, either directly, by policy trading or negotiation. Bindings are set up by pluggable binder components. We also introduce a way to open up middleware engineering through PPI's (policy programmer interfaces). A proper hierarchy for PPI's supports pluggability of policies.

The distinction between bindings and their activations allows reasoning about bindings with changing activations and activation policies and policies for managing activations as separate entities.

Our approach to the second problem area is to introduce a model for stating non-functional (QoS) requirements and environmental properties. In essence this is a type/subtype model, i.e. a user requires a specific type of QoS and the policy requires a specific type of service from the environment. A subtyping relationship ensures that the requirement can be met by a compatible offer, typically by using a special trading service, possibly combined with client/server negotiation. Metapolicies which involves policy-trading or negotiation can then map from declarative QoS descriptions to suitable policies.

3. Results and further work

We have developed a model for stating requirements/environmental properties and demonstrated how this support automatic policy selection/negotiation (and admission control) when binding to object interfaces. We demonstrated how the QoS model and policy-trading can be applied to an application [1].

We have designed and implemented middleware support (The FlexiBind framework) for flexible bindings, based on a distinction between binding establishment and binding activation [2]. The prototype supports policy-governed binding, binders, activators and PPIs, and is realised as an extension of the ANSA FlexiNet ORB framework [7].

Current work involves experiments with policy-trading and negotiation-protocols based on the QoS model and the FlexiBind framework. Investigate how our model and framework meets the requirements of multimedia applications. This involves experiments using stream interfaces.

References

1. Ø. Hanssen, F. Eliassen, "Towards a QoS aware Binding Model", Proc. SYBEN'98, Spie Press, May 1998.
2. Ø. Hanssen, F. Eliassen, "A Framework for Policy Bindings", To appear in proc. DOA'99, IEEE Press, September 1999.
3. G. Kiczales, "Beyond the Black Box: Open Implementation", IEEE Software, 1996, 13(1), p. 8-11.
4. J.A. Zinky, D.E. Bakken, R.E. Shantz, "Architectural Support for Quality of Service for CORBA Objects", Theory and Practice of Object Systems (Special Issue on OMG and CORBA), January 1997.
5. G. Kiczales et. al., Aspect-Oriented Programming", ACM Computing Surveys, 28 (4es)
6. G. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. Middleware '98, Springer Verlag, 1998
7. R. Hayton et. al. "FlexiNet Architecture Report", ANSA Phase III report, February 1999

Policy Trading

Øyvind Hanssen*, Frank Eliassen†

*Agder University College, Faculty of Engineering, Grimstad, Norway
†University of Oslo, Department of Informatics, Oslo, Norway
ohanssen@acm.org, frank@ifi.uio.no

Abstract:

In this paper we investigate policy trading to address the problem of how extensible and configurable middleware could adapt to different non-functional requirements and different properties of environments. Policies denotes potential contracts between the system and the user, i.e. if requirements for the environment is met, the policies guarantees that certain QoS will be provided.

Trading involves matching user requirements plus environmental properties with policies. In this paper we define policy trading and illustrates its principles and use. Trading is based on profile expressions which are references to statically defined profile graphs. Such expressions can be combined through simple adding (sum) or adding expressions which refers to different locations (sidesums). We develop a set of rules to allow for testing of compatibility relationships between profile expressions.

1. Introduction

There is much interest in middleware that support automatic adaptation to different and changing non-functional application requirements and environmental properties. Much of the research in this area has been motivated by the need for distributed multimedia support and much attention has been paid to bindings between continuous media stream interfaces which need resource management to meet different Quality of Service requirements. Many of those concepts are also useful for *operational services* where applications may have requirements for binding (e.g. reliability, security and performance), especially in the area of mobile client/server systems [1] where connectivity and resource availability may change dramatically over time.

An example of an type of application which may benefit from adaptability to QoS requirements and environmental properties, especially when dealing with mobility, is command and control systems like described in [2]. For military or emergency traffic, email messages could be associated with QoS requirements (e.g. delivery time, security, reliability) and different delivery methods might be selected according to the requirements and what the environment and network is capable of [3].

Our goal is to develop the foundations of a *QoS aware binding facility* that can automatically map from application requirements plus a description of the available system properties to a suitable binding. This binding should also be able to adapt to changing system properties by changing its protocols, resource policies etc. It is possible to view the configuration of a binding as a result of a policy. A policy is designed to meet application requirements for a given set of environmental properties (resource availability, transport QoS etc.).

In previous work we outline and advocate the use of the policy concept and a model for stating properties and selecting suitable policies by *policy trading* [3]. In [7] we design middleware support for adaptable policy governed bindings. In this paper we focus at the foundations of the automatic selection of policy components to be deployed in heterogenous environments participating in a binding, i.e. policy trading. To allow this we need a model which allow stating requirements and environmental properties clearly and which allow automatic checking if the environments (through the policy) satisfies the requirements. In principle, policy trading should be regarded as a simple function like illustrated in figure 1. A trading function selects a policy that can meet the application requirement in the given environment.

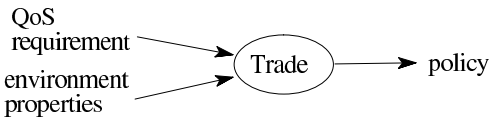


Figure 1. Simple view of policy-trading

This is sufficient if we could view a binding as a single entity, running in a single environment and with a single QoS requirement. But in practise we deal with different locations, with heterogeneous environments. For instance the client and server side may be running on different platforms offering different environmental properties which are not known statically. The different sides may have different application QoS requirements for the same binding. Here, policy trading seems to be useful, but we need to combine the requirements and the environmental aspects of the different sides and trade two (or more) matching policy components instead of one like in the simple case.

To address the above issues we define a type model for requirements and environmental properties. Such types are called 'profiles' and are used to describe the potential contract offered by policies and to state application requirements and environmental properties when trading for suitable policies. A *subtype relationship* between profiles and combinations of profiles are the foundations for the trading of policies.

The rest of the paper is structured as follows: In *section 2*, we define the semantics of policy trading. We introduce the idea of policies as mappings from environments to the satisfaction of user requirements, as potential QoS contracts, and as software components which establish bindings. We define the meaning of locations, how to combine requirements from different sides and how to combine environmental descriptions at different locations. We briefly define single side and composite (more than one side) policy selection.

In *section 3* we define a model of profiles as the abstraction we use to state user requirements and environments. In general profiles are expressions which are either (1) basic profiles which are nodes in a type/subtype graph where the subtype relationships are explicitly defined, (2) sums of profiles which in essence are combination of basic profiles at the same location, or (3) sidesums which in essence are combination of basic profiles at different locations. We develop rules for testing if there is a satisfaction relationship between two arbitrary expressions.

In *section 4* we illustrate policy trading briefly by providing an example. *Section 5* is a brief discussion on how policy trading relates to extensible middleware and how such middleware might use it for selecting and installing policy components. We compare our work with related research in *section 6* and in *section 7* we conclude.

2. Policy trading model

We introduce the concept of *policy trading* to facilitate the selection of policies for creating bindings that match user requirements and the capabilities of the environments the bindings will run in. Policy trading is inspired by ODP trading [8], but instead of functional interface references we trade policy implementations to be used to set up bindings. Instead of interface types and property sets, we use *user profiles* (non-functional properties of resulting bindings) and *service profiles* (requirements to environments applications are running in). In practical ODP traders (c.f. ANSA/CORBA) the use of type-graphs makes ODP trading effective, a similar approach to service- and user- profiles should work for policy trading.

2.1. Policy

A *policy* can be viewed as a mapping from a set of environmental properties S to a user level quality of service (or satisfaction of an user requirement).

$$p{:}S \rightarrow U$$

The *user profile* U denotes the satisfaction of a non-functional user requirement R. This is properties of the binding as perceived by users or applications (Quality of Service). An example may be that the binding guarantees a certain level of confidentiality of the exchanged information. A policy which satisfies such a requirement typically involves encryption.

The *service profile* S denotes a predicate on the environment E. If S is satisfied by E, the policy will deliver U which satisfies R. Service profiles may be statements about availability of services or resources which supports the engineering of bindings. For instance, a policy may require the ability to set up network connections with a certain QoS level or the availability of the secure socket layer (SSL) or a certain amount of available buffers.

A policy denotes a potential contract between the system and potential users. If a requirement to the environment is satisfied, the policy guarantees that certain properties will be true. If $P(x)$ is the predicate defining a profile x, a policy states the following:

$$P(S) \Rightarrow P(U)$$

A policy also denotes a way to enforce its contract, i.e. a configuration of resources and mechanisms. We refer to this as the *policy-implementation.* In practise this could be a software component which is executed to do the configuration (activation of a binding), for instance a Java class.

2.2. Location and satisfaction relationships

Since we are considering bindings between components in open, distributed systems it is necessary to consider *location* in the sense that different sides of a binding may have different application requirements and offer different environments. For instance a server may require that all bindings to it are transactional and the client may require a certain minimum response time. A policy for the binding must satisfy both of these requirements. On the other hand the client and the server may run on different machines, with different operating systems and/or middleware to support the bindings. A policy may for instance require access to stable storage at the client side and a certain amount of free memory at the server side.

Here (and in the rest of this paper) we limit our discussion to the client/server model but we envisage our ideas should be applicable more generally too (for instance multiparty bindings). It is an issue for future research to see how our trading model applies to non-client/server models.

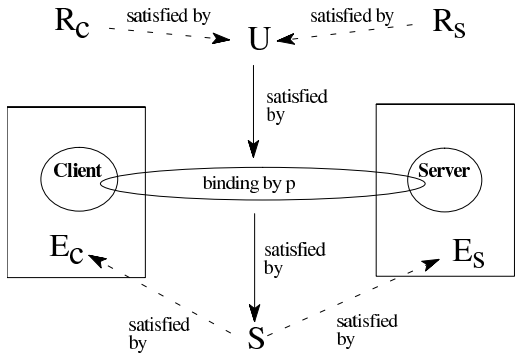


Figure 2. Client/server bindings and satisfaction relationships

Combining requirements. Consider figure 2 above where U is the user profile and S is the service profile of a policy p. Furthermore R_C and R_S denote the user requirements of the client and server respectively. In order to use p as the policy for the binding, U must satisfy the combination of the requirements R_C and R_S. In order to express combined requirements like this we introduce the '+' operator such that the combined requirement is expressed as $R_C + R_S$. The fact that U satisfies the combination R_C and R_S, we thus express as follows (the satisfaction relationship is denoted '≤'):

$$U \leq (R_C + R_S) \;\Leftrightarrow\; U \leq R_C \wedge U \leq R_S$$

Combining environments at different locations. The environment E is determined by the location of each component to be bound. The environment for a potential binding is thus a combination of the environments of the locations of the participants. So, the service profile S of a policy must be satisfied by both sides of a binding, i.e. it must be mapped to requirements to the actual locations. S holds if satisfied by a combination of E_C and E_S where E_C and E_S are the environment of the client and server respectively (see figure 2). In order to express combined environment-descriptions at different locations like this, we introduce the '⊕' operator. The fact that the combination of E_C and E_S satisfies S we thus express as follows:

$$(E_C \oplus E_S) \leq S \;\Leftrightarrow\; E_C \leq S_C \wedge E_S \leq S_S$$

$$\textit{where}\;\; S = (S_C \oplus S_S)$$

Note that the '⊕' operator is different from the '+' operator in the sense that each side of it refer to different locations (or sides of the binding). Each side is treated independently with respect to the satisfaction relationship. This also means that a policy can consist of two parts where one part is for the client side and one is for the server side. Each of these has a separate requirement for the environment, hence we can split S into S_C and S_S. The meaning of the '+' and the '⊕' operator in profile expressions is defined further in section 3.

2.3. Trading architecture

Policy trading is the process of finding a policy whose user profile and service profile matches the requirement R and the environment E. Trading man be viewed as a mapping *trade* from a user requirement R and an environment E to a policy P:

$$\textit{trade}: R \times E \rightarrow p$$

Like in ODP trading, two kinds of operations are important in policy trading: (1) *Export* which registers a policy with the trading service and (2) *import* which returns a policy (maps directly to the trade primitive above). Generally, policies for client/server bindings consists of two components and we need to trade policy-components for both sides of the binding. Those two components may be traded separately or as a whole (as a single policy). Therefore we distinguish between single side and composite policy selection:

Single side policy selection. In some cases we trade at one side only (typically the client side). The policy at the other side is fixed and known (possibly traded separately at an earlier stage). The result of the server side policy (e.g. what protocols it supports) may be part of the environment of the client (E_C). We can describe single side policy selection, *Trade* $(U + U_C,\; E_C)$ as follows:

$$\text{Find a policy } p{:}S \rightarrow U \text{ such that } E_C \leq S \wedge U \leq R_C + R_S$$

Composite policy selection. Generally, we trade tuples $<p_1, .. p_n>$ where each p_i is a policy to be used at a side i. In the client/server model we can describe composite policy selection, *Trade*$(U_S + U_C,\; E_C \oplus E_S)$ as follows:

$$\text{Find a policy pair } (p_c{:}S' \rightarrow U',\; p_s{:}S'' \rightarrow U'')$$

such that

$$E_C \oplus E_S \leq S' \wedge U' \leq R_C + R_S$$
$$E_C \oplus E_S \leq S'' \wedge U'' \leq R_C + R_S$$
$$p_c \text{ is interoperable with } p_s$$

The last requirements (p_C is interoperable with p_S) means that the policy components at each side must be able to interoperate with each other to fulfil their task, i.e. they use compatible protocols. In this paper we assume that each such tuple (typically client/server pairs) is specified manually when exporting policies, i.e. a policy tuple is treated as one single policy also at export time. It is an issue for future research how to automatically match such tuples at import time.

3. Profile model

In our proposed profile model, non-functional (QoS) requirements are specified as *user profiles* while the computing and networking environment are described as *service profiles*. Policies are related to user profiles through a satisfaction relationship, and to service profiles through a requirement relationship. Hence a specific policy satisfies an user profile (the qualities it provides) and requires a service profile (the resources it requires from its environment). Furthermore, there is a compatibility relationship between profiles (subprofiles). In this section we define the semantics of basic profiles as nodes in a graph and how we can state profiles as expressions by combining basic profiles with the operators '+' and '⊕'. Furthermore we define subtype relationship between such expressions. These will be used when developing rules for testing compatibility between arbitrary profile expressions.

3.1. Basic profiles

A basic profile *X* has a name and denotes a predicate *P(X)* (which may be implicit). A requirement may then be stated as a reference to the profile by name. Implicitly this means that the requirement is that the predicate should evaluate to true.

The predicates denoted by profiles are usually not directly evaluated or even stated explicitly at all. The idea is to use profile names and subprofile relationships between them to test if a profile offer matches a profile requirement. However, policy implementers may need to know the exact meaning of profiles. Subprofile relationships are established explicitly (c.f. types and subtype-relationships in the ANSA trader).

Figure 3 below shows a simple example of a user profile graph for an email application (see [3]). Users can specify requirements for message delivery which are mapped to this graph. For instance, '*Authenticated*' is a subprofile of '*Secure*', i.e. it includes the requirements of '*Secure*'.

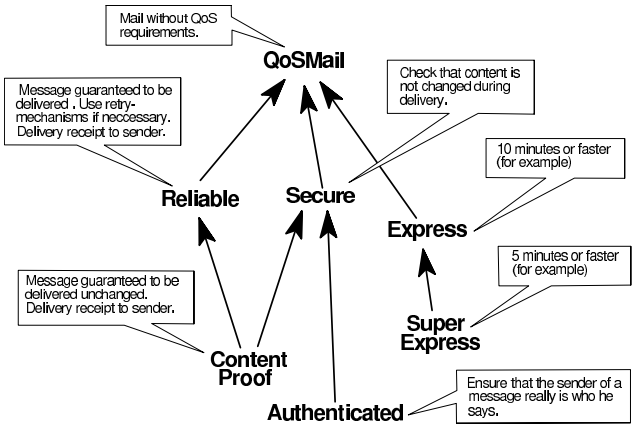


Figure 3. Example profile graph

We define a *subprofile relationship* as follows. A subprofile *Y* of *X* is compatible with *X* but may be a stronger requirement. Thus a requirement stating a profile *X* is satisfied by any profile which is subprofile of *X*:

$$Y \leq X \Leftrightarrow \left(P(Y) \Rightarrow P(X)\right)$$

Since the implication relationship is transitive the subprofile relationship is transitive.

3.2. Profile sums

While profile graphs (explicitly defined profile-names and subprofile relationships) should be statically defined and standardised, it is also useful to allow anonymous profiles to be defined dynamically as sums of basic profiles from profile graphs. We define an addition operation for profiles as follows:

$$P(X+Y) \Leftrightarrow P(X) \wedge P(Y)$$

It should be clear that $X+Y$ is subprofile of X and of Y (c.f. section 2.2.1). Further, we make the following observation: Given a profile graph where $C \leq A$, $D \leq B$, $E \leq C$ and $E \leq D$. Then E is a subprofile of $A+B$. By the definition above and the principle of transitivity it follows from the subprofile relationships that:

$$P(E) \Rightarrow P(A) \quad and \quad P(E) \Rightarrow P(B)$$

From this it follows that:

$$P(E) \Rightarrow P(A) \wedge P(B) \quad \Leftrightarrow \quad E \leq A+B$$

The observations above can be generalised into a rule for testing subprofile relationship between sums. First, a given basic profile is a subprofile of a sum if and only if it is a subprofile of all elements of the sum:

$$y \leq \sum_{i=1}^{n} x_i \quad \Leftrightarrow \quad \forall x_i \in \{x_1 \ldots x_n\} : x_i \geq y$$

and furthermore, a sum is a subprofile of a given profile Y if we can find some part of the sum that is subprofile of Y. If Y is restricted to be a basic profile the subprofile relationship is satisfied if and only if at least one of the sum's elements are subprofile of Y:

$$y \geq \sum_{i=1}^{n} x_i \quad \Leftrightarrow \quad \exists x_i \in \{x_1 \ldots x_n\} : x_i \leq y$$

From these two rules we can deduce a general rule for subprofile relationship between sums of basic profiles:

$$\sum_{i=1}^{n} x_i \leq \sum_{j=1}^{m} y_j \quad \Leftrightarrow \quad \forall y_j \in \{y_1 \ldots y_m\} : \left(\exists x_i \in \{x_1 \ldots x_n\} : x_i \leq x_j\right)$$

3.3. Profile sidesums

If we are trading *composite policies* (c.f. section 2.3.2), each exported composite policy should have a service profile requirement that combines properties of local environments of both (all) sides participating in the binding.

In section 2.2.2. we introduced the '⊕' operator (an expression constructed with '⊕' is called a '*sidesum*') to allow us to combine statements about different locations. To see how this relates to graphs of basic profiles, consider the service profile graph in figure 4: A composite policy which requires a client environment E_1 and a server environment E_2 is exported to the trader An import is attempted with the client environment E_4 and server environment E_3. If client and server locations are identical, the '+' operator can be used to combine profiles, hence $S=E_1+E_2$ and $E=E_4+E_3$. In this case there will be a match if $E_3<E_1$ and $E_4<E_2$ (c.f. figure 4). In this special case, a sidesum is equivalent to a sum.

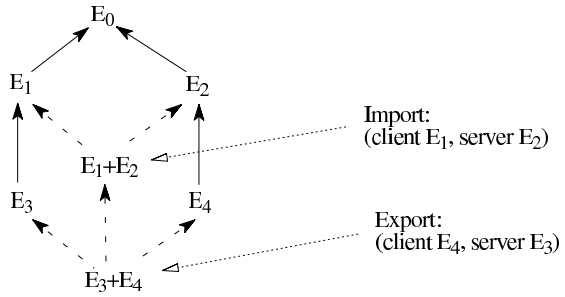


Figure 4. Sum for multisided environment

However, if we combine statements about *different* locations, this would be wrong. Figure 5 below illustrates the meaning of the '⊕' operator: The basic profile graph is applied separately to each location involved such that $E_1 \oplus E_2 \Leftrightarrow E'_1+E''_2$. We observe that now there will be no match for the example used here.

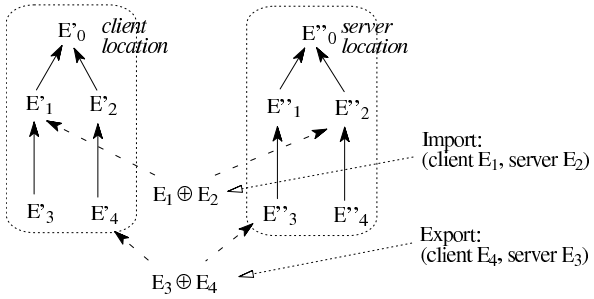


Figure 5. Sum for multisided environment

The rule in section 3.2. that defined the subprofile relationship for sums could be modified for sidesums. It should not be difficult to see from the discussion here and in section 2.2.2. that a subprofile relationship between two basic profiles can only exist if they are at the same location: $X \leq Y$ if and only if $loc(X)=loc(Y)$.

$$\Phi_{i=1}^{n} X_i \leq \Phi_{j=1}^{m} Y_j \Leftrightarrow$$

$$\forall Y_j \in \{Y_1 \ldots Y_m\} : \left(\exists X_i \in \{X_1 \ldots X_n\} : (loc(X_i) = loc(Y_j) \wedge X_i \leq Y_j) \right)$$

$$\text{where } \Phi_{i=1}^{n} x_i = x_1 \oplus x_2 \oplus \ldots \oplus x_n$$

Now, we still need a way to determine if two elements of two different sums refer to the same location. The simplest approach is to assume that order of the elements of a sidesum expression matters and that $loc(X_i)=loc(Y_j)$ if and only if $i=j$. This makes implementation simple, but we have no way to determine which side i is referring to (client or server). An ad. hoc. way to solve this problem is to make one or more of the locations a node in the profile graph. An expression can then look like for instance $(E_1+'server') \oplus E_2$. Now, it is easy to determine that E_1 is about the server side and that E_2 is about the client side.

3.4. Practical testing of subprofile relationships

A practical implementation of a policy trading service will be based on a way to determine subprofile relationship between two arbitrary profile expressions, which each may be a combination of sums and sidesums.

First, a trading service needs to store information about *profile graphs* (c.f. type graphs in ANSA/ODP trader). Such graphs can be regarded as static information. They are typically agreed on within a domain and seldom changed. The complexity and scalability of using such graphs is a case for research, but a typical starting point is that each application or application domain defines a pair of graphs and that federations could be built between different domains (like in ODP trading). The hierarchical structure naturally support federation and extensibility.

Policy traders must store two separate graphs: The user-profile graph and the service profile graph. Furthermore, the trader should store information about the *policies*. Each policy contains a user-profile and a service-profile which are profile expressions.

Those type-graphs will be referenced in *profile expressions*, which denotes profiles as sums or sidesums of profiles which may be single references to nodes of the type-graphs or sums/sidesums recursively. In practise, expressions are represented as parse trees. Figure 6 illustrates how we could represent a profile expression:

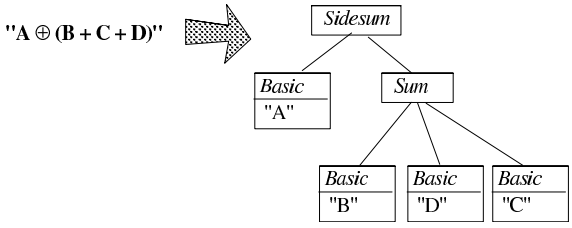


Figure 6. Parse tree for profile expression

Testing subprofile relationship between single nodes in profile graphs is straightforward: It is a classic graph search (DFS or BFS) for reachability between nodes. A subprofile test between expressions is somewhat more complicated. In section 3.2. we define a formula for testing two sums and in section 3.3. we modify it for testing two sidesums. We may also need to test expressions of different types. Here we do the following observations:

- A sum and a basic profile expression are compared by treating the simple expression as a sum containing only one element. The same can be done for sidesums, but if super profile expression has only one element, the sum test and the sidesum test are equivalent.
- It follows from the formula from section 3.2. that a subprofile sidesum must have at least as many elements as the superprofile sidesum. Therefore, a basic profile can never be a subprofile of a sidesum, but the opposite is possible.
- For the same reason, a sum can never be a subprofile of a sidesum, but the opposite is possible. A sum may be viewed as a node in a single graph instance.

This reflect the fact that a multisided policy need a multisided environment. A policy containing *n* side-components need an environment with at least *n* sides. We summarise these observations in figure 7. Testing any expressions can be decomposed into simple tests (graph search), sum tests or sidesum tests.

SUPER \ SUB	**Simple**	**Sum**	**Sidesum**
Simple	Simple graph search	Sum test	Sum test
Sum	Sum test	Sum test	Sum test
Sidesum	FALSE	FALSE	Sidesum test

Figure 7. Profile expression subtype test

4. Example

In [3] we use a QoS aware email application as a case to demonstrate the usability of policy trading. We can select from a set of delivery policies depending on what requirements the user has for delivery (fast, reliable etc..) and the capabilities of the channel to be used for delivery. The trading process also include selection from alternative channels. Trading is simple in this example. Service profiles describe channels and the application do not need to take different sides into consideration.

Here, we illustrate multisided binding in a client/server architecture by showing how we could select security policies. Note that this is not meant to show how we could provide a good security architecture, but just to illustrate our idea of policy trading.

First we have a user profile graph defining different types and levels of security, for instance authenticity and secrecy or combinations. Secondly, we have a service profile graph defining if the client or server side environments offer the secure socket layer (SSL), secure storage for keys, high performance CPU or access to some kind of certification authority.

Third we have a set of composite (double sided) policies, providing security of different types (by using different encryption techniques and/or protocols for authentication or key exchange). Figure 8 shows a set of policies and their relationships with profile graphs. Note that one side in a sidesum may indicate that it is meant for the server role by adding a node in the graph:

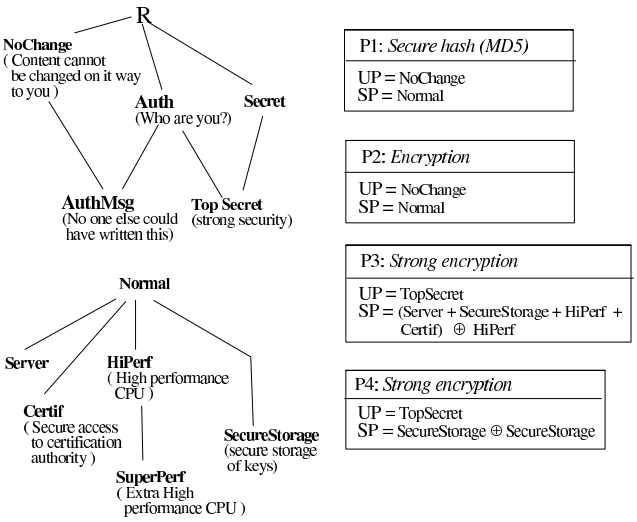


Figure 8. Example of profile graphs and policies

If the client wants a "*Secret*" binding, the server wants an "*Auth*" binding, the client environment offer "*SuperPerf*"and the server environment offer "*Certif*", "*SecureStorage*" and "*SuperPerf*", the invocation of the trader could go like this:

trade ("Secret + Auth",
"(Server+Certif+SecureStorage+SuperPerf) ⊕ SuperPerf")

The trader should now select only policy P_3. "*TopSecret*" is the only user profile offered, which is compatible with both "*Secret*" and "*Auth*". Furthermore, the environments are compatible with what the policy requires. Policy

P_4 should not be selected because only one side has "*SecureStorage*" and P_4 require "*SecureStorage*" at both sides. Figure 8 below illustrates how policy P_3 satisfies requirements of client plus server and how client and server environments satisfies requirements of the policy.

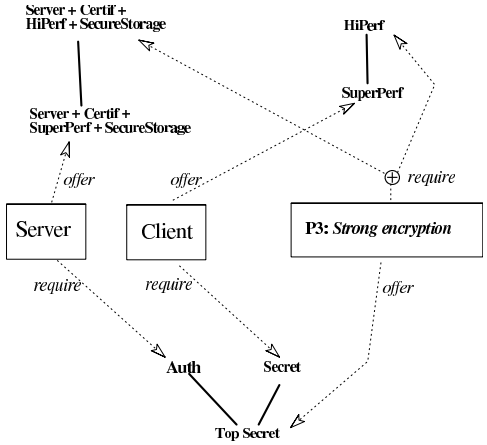


Figure 9. Matching requirements and offers

5. Policy trading in middleware

Policy trading can be used in component based middleware to select policy components for bindings. Many approaches is being developed for designing such middleware: Authors has applied the concepts of reflection [19] and aspect orientation [6] to middleware to address extensibility and configurability (i.e. to open it up) without destroying the benefits of object oriented abstraction. The hypothesis is that there should be a separation of concerns between functional and non-functional aspects of bindings.

In our context this means that the non-functional aspect is dealt with by *policies*. A policy is in our approach represented by a software component, which can be plugged into the middleware and which is responsible for configuring the binding. In [7] we describe such a middleware framework, based on Java and the ANSA FlexiNet framework [20]. Here are some highlights:

- Bindings are configured (activated) by pluggable and replaceable *activator* components. An activator represents a policy.
- Bindings are established (but not necessarily activated) by pluggable *binder* components. A binder represents a *metapolicy* which governs how policies are managed, for instance how activators are selected and how bindings could be adapted by replacing activators.
- Policy components are given access to the facilities of the middleware platform through *policy programmer interfaces* (PPI's).

  Different ORB's (object request brokers) may offer different PPI's. There is a type/subtype hierarchy for PPI's. A policy component will require a PPI type and if the ORB offers a subtype of it the policy can be used.

Metapolicies may involve policy trading to select activator components. Furthermore PPI types should be mapped more or less directly to service profiles when trading. PPI's covers the static aspects of service profiles, but not the dynamic aspects such as for instance resource availability and QoS of the services offered. These could be results of measurements and/or monitoring and given as feedback to metapolicy objects to trigger adaptation.

5.1. Binding protocol issues

When different locations are involved in binding there is a need for a protocol to collect requirements and environmental descriptions and to safely install policy components at each side participating. When looking at architectures for binding, an issue is *where* to do the trading. This could be done at the client side or at the server side. Here we briefly discuss two possible architectures for binding which are likely to be used.

Policy trading at client side. In this architecture there is only one policy to select and it is selected (traded) at the client side when binding to a (remote) interface. The server has a policy installed *statically*. The policy of the server interface limits what could be used at the client side. For instance, a server policy may include concurrency control and logging (for recovery) and require bindings to it to be transactional. Exported interface references should if necessary indicate such server imposed limitations, which must be taken into account when trading for a policy. I.e. it might be used as part of the environment expression.

The traded policy may include (partly) configuration of the server side, i.e. the policy installs a component at the server.

Policy trading at server side. An alternative is to do the trading at the server side. In figure 10, we show how this could be done by a scenario where composite policy-trading (c.f. section 2.3.2.) is used. First (1), the client invokes a *bind* operation where user- and service profile are arguments, at the server. The result of this operation contains the policy. The server may also have some requirement (user profile) and it finds a user profile which is a subprofile of both user profiles (i.e. the sum). The server then invokes the trading service (2) to find matching policy pairs. It installs the server component of the result (4) before it returns the client component to the client (5) which then installs it (6).

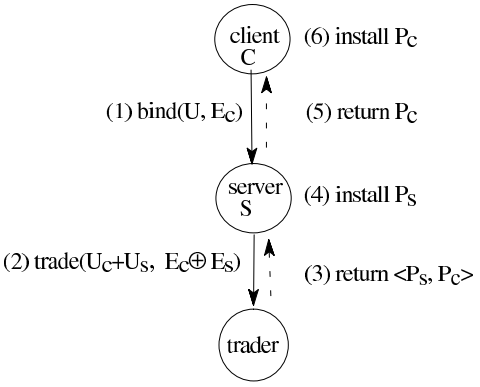


Figure 6. Server trading binding scenario

6. Related research

Our work builds on the idea of QoS provision and configurability of middleware via policies, which is not new. For instance the Open OODB project [10] aims to develop an architecture for extensible database middleware. Here, operations like object access or selection may be associated with invariants (requirements), which can be satisfied by different policies. Policies are realised by policy-performers and managed by policy managers (hides the choice of policy). Compatibility between policies and between the properties they require from the environment was studied in [11], including composition of properties. Our approach is a practical application of some these ideas in a specific problem domain. However, we talk about compatibility between the QoS they provide, not about compatibility between the policies themselves.

Trading [8] is a well known approach to the selection of services (or computing resources) to bind to. We adopt the concept of trading, but in our context we do not trade the services themselves, but we trade *QoS contracts* and implementation policies to be used for *bindings* between components (typically between services and their clients). The ideas of types and subtypes [12] can successfully be applied to QoS requirements and the properties of environments. Our notion of types do not capture operational (or functional) behaviour, but non-functional properties like QoS and/or resource availability in the engineering viewpoint. Furthermore, the meaning of types are implicit and the relationships are explicitly established like in the ANSAware trader [13]. An important difference from ODP trading is that we use *two* different type graphs. One which captures the provided QoS and one which captures what the policy needs from the environment to be able to fulfil its contract.

Much work has been done on QoS in the context of Multimedia support in the RM-ODP [14] and *QoS architectures* (e.g. [15, 16, 17]) which look at models for describing QoS requirements, binding establishment which include admission control, negotiation and adaptation. For instance the QoS broker architecture [17] facilitates negotiation between application and system, mapping of QoS parameters and orchestration of resource management at different levels (network, hosts, O.S. etc.). Our model of policies capture the resource management which may include all aspects of a binding. The process of selecting and installing policies can be related to the QoS brokerage model but many of the mechanisms of the QoS broker entity will be policy specific in our approach. We performed a survey on QoS research in a separate report [18].

In our model, negotiation and admission control is all done by policy-trading, i.e. multisided trading finds a QoS contract which satisfies both (or all) parties. Trading will only return contracts which are satisfied by the actual environmental properties, hence trading realises admission control. Transparent adaptation or renegotiation may be initiated if the environment changes its service profile. Bindings may be associated with adaptation policies which monitors the environment and initiates *re-trading*. If this cannot return a contract which still meets the agreed user profile, this must be renegotiated with the application.

Our contractual approach to QoS management is inspired by the QuO architecture [5]. QuO extends the functional interface definition language (CORBA IDL) with a QoS description language (QDL) which captures application's expected usage patterns, QoS requirements and resource usage for bindings. Our profile concept is inspired by the QuO project's concept of *negotiated* and *reality regions* (ranges of acceptable non-functional values), but it is more directed towards establishing compatibility relationship graphs through the notion of type inheritance rather than specifying meaning of each region by listing value ranges.

Much research is also done in the problem area of (re)configurable and extensible middleware. There is currently a growing interest in aspect oriented or reflective middleware. For instance, the Lancaster University looks into at how to apply the concepts of reflection and components to middleware [19] and especially at how to *open up* the engineering of bindings [9] in a collaboration with Norwegian researchers [21].

Our approach can be regarded as aspect oriented [6] in the sense that we distinguish between functional and non-functional aspects of a binding. This also follows from the open implementation approach, [4] which proposes that non-functional issues should be dealt with separately from functional issues and exposed to programs in separate interfaces (Meta Object Protocols). The non-functional aspect is programmed separately from functional interactions and managed as policies. Aspect weaving should also be done dynamically to support adaptability. The QuO architecture also adopts this approach to allow object designers to expose key design decisions that affect Quality of Service. This makes it possible to alter the non-functional behaviour of distributed applications by choosing the implementation which is best suited for the situation. Non-functional aspects (contracts) are here specified in a special contract definition language (CDL). In our approach, potential contracts are defined by defining profile graphs and exporting policies to the trader.

7. Conclusions

In this paper we apply the idea of trading to map from QoS requirements and properties of environments to suitable policies for binding. The process of trading represents both negotiation, admission control and mapping of QoS. This is different from ODP trading in the sense that we do not trade services to bind to, but rather how to bind to achieve certain non-functional properties of bindings.

We have developed a model of policy trading as a way of matching both user requirements and environmental properties with policies. *Policies* are potential contracts between the system and the user. If a requirement for the environment (service profile) is met, a policy guarantees that a certain QoS will be provided (user profile). Policies also denotes a implementation and policy implementations are typically small software components which can activate and configure the bindings such that the requirements are met. The model also deals with the fact that the properties used for trading reflect the requirements and environments of both sides of the binding, thus we must be able to combine such descriptions. For environmental properties, each side must be treated separately with respect to the satisfaction relationship.

We develop a model of *profiles* as a means to state requirements and environmental properties. This model defines basic profiles as nodes in statically defined directed acyclic graphs where edges explicitly define subtype relationships. Hence, the meaning of the profiles could be implicit. A simple notion of adding profiles (sum) allows us to combine basic profiles, which represent different aspects of requirements or environments. Furthermore, a notion of adding profiles at different locations (sidesum) is developed. Basic profiles combined with sums and/or sidesums form *profile expressions*. We also provide a means for testing subtype relationships between such expressions.

We have shown how trading can be used to find the correct policy for a binding and how it can be used to find matching pairs of policies to be used in heterogeneous client/server environments. We also discussed the how a practical policy-trader works by documenting the essentials of the trading and matching algorithms.

8. Acknowledgments

An early phase on the work described here was performed when the author was seconded by the University of Tromsø to the ANSA Phase III programme at APM Ltd. (now Citrix) in Cambridge UK in 1997. This was supported by a NATO Science Fellowship through the Norwegian Research Council grant no. 116590/410. Thanks to Andrew Herbert, Richard Hayton and the rest of the ANSA team for valuable comments. Also thanks to Gordon Blair and the other researchers from the universities in Lancaster, Oslo and Tromsø who was involved in the CORBAng project in 1998 and 1999. Also thanks to Gregor Kiczales and the other mentors and Ph.D. students at OOPSLA'99 Doctoral Symposium.

9. References

[1] J. Jing, A. Helal, A. Elmagarmid, "Client-Server Computing in Mobile Environments", ACM Computing Surveys, Vol. 31, No. 2, June 1999.

[2] J.Ø. Aagedal, Z. Milosevic, "Enterprise Modelling and QoS for Command and Control Systems", 2nd International Enterprise Distributed Computing Workshop (EDOC'98), November 1998. .

[3] Ø. Hanssen, F. Eliassen, "Towards a QoS aware Binding Model", Proc. SYBEN '98, Zurich, May 1998.

[4] G. Kiczales, "Beyond the Black Box: Open Implementation", IEEE Software, 1996, 13(1).

[5] J.A. Zinky, D.E. Bakken, R.E. Schantz, "Architectural support for Quality of Service for CORBA Objects", Theory and Practise of Object Systems (Special issue on the OMG and CORBA), January 1997.

[6] G. Kiczales, J. Irwin, J. Lamping, J-M. Loingtier, C.V. Lopes, C. Meada, A. Mendhekar, "Aspect Oriented Programming", ACM Computing Surveys, December 1996.

[7] Ø. Hanssen, F. Eliassen, "A Framework for Policy Binding", Proc. DOA'99, Edinburgh, IEEE Press 1999.

[8] M. Y. Bearman, "ODP Trader", Proc. ICODP'93, Berlin 1993, pp. 19-33.

[9] T. Fitzpatrick, G.S. Blair, G. Coulson, N. Davies, P. Robin, "Supporting adaptive multimedia applications through open bindings", Proc. ICCDS'98, May 1998.

[10] D.L. Wells, J.A. Blakeley, C.W. Thompson, "Architecture of an Open Object-Oriented Database Management System", IEEE Computer, October 1992.

[11] H.M. Hinton, E.S. Lee, "The Compatibility of policies", Proc. 2nd. ACM Conference on Computer and communications security, November 1994.

[12] J. Indulska, M. Bearman, K. Raymond, "A Type Management System for an ODP Trader", Proc. ICODP'93, Berlin, September 1993.

[13] R. van der Linden, J. Sventek, "The ANSA Trading Service", IEEE Distributed Processing Technical Committee Newsletter, Vol. 14, No. 1.

[14] G. Blair, J.B. Stefani, "Open Distributed processing and Multimedia, Addison Wesley, 1997.

[15] C. Aurrecoechea, A.T. Campbell, L. Hauw, "A Survey of QoS architectures", Multimedia Systems Journal, Special issue on QoS architecture, 1996.

[16] C.A. Nicolaou, "A Distributed Architecture for Multimedia Communication Systems", Ph.D. thesis, Computer Laboratory, University of Cambridge, 1991.

[17] K Nahrstedt, J. Smith, "The QoS Broker", IEEE Multimedia, spring 1995.

[18] Ø. Hanssen, "FlexiNet - QoS Investigation", ANSA Phase III Technical Report 1977.01.00, June 1997.

[19] G. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. Middleware '98, Springer Verlag, 1998.

[20] R. Hayton et. al. "FlexiNet Architecture report", ANSA Phase III report, Feburary 1999

[21] F. Eliassen, A. Andersen, G.S. Blair, F. Costa, G. Coulson, V. Goebel, Ø. Hanssen, T. Kristensen, T. Plagemann, H.O. Rafaelsen, K.B. Saikoski, W. Yu, "Next Generation Middleware: Requirements, Architecture, and Prototypes", Proc. FTDCS'99, IEEE Press 1999.

Towards Declarative Characterisation and Negotiation of Bindings

Øyvind Hanssen
University of Tromsø
Department of Computer Science
9037 Tromsø
+47 95117457
ohanssen@acm.org

ABSTRACT

This paper addresses negotiation of bindings in open systems, and in particular how to characterise the capabilities of heterogeneous platforms, and communication channels. Based on a middleware architecture supporting policy-governed binding, negotiation is about selecting suitable policies for bindings at run-time. We propose a model for declarative expressions based on a hybrid of declared and rule-based conformance, and composition operators. We also propose a scheme for how the middleware can support automatic characterisation of resources or other relevant capabilities and composition of these, based on the declarative expression model.

Categories and Subject Descriptors

D.2.1 **[Software Engineering]**: Requirements/Specifications -- *languages*. D.2.12 **[Software Engineering]**: Interoperability -- *distributed objects*.

General Terms

Experimentation, Languages.

Keywords

Quality of Service, Negotiation, Binding, Trading, Middleware.

1.INTRODUCTION

In the last decade much attention has been turned towards middleware which supports dynamic adaptation to non-functional application requirements and varying environmental conditions. This is motivated by requirements for e.g. multimedia applications, mobility, dependability, etc. The capabilities of platforms on which to build open and distributed applications are also increasingly diverse. Platforms may offer different types and amounts of resources for computing and communication, as well as different mechanisms to manage them. The approach of middleware providing one single abstraction, hiding implementation details and differences between the various platforms is recognized to be too limiting. Therefore, research has been focusing on opening up and componentising the middleware, to make it more configurable, but still without sacrificing abstraction.

RM'05, November 28–December 2, 2005, Grenoble, France.

Reflective middleware [1] explores the idea of using meta-level architectures for exposing implementation details, and using meta object protocols for programmatic access to these. It is however less clear how to support automatic adaptation to various QoS requirements and environmental properties. Binding between components would involve a negotiation process, which involves exchanging requirements and offers, to reach agreement on a contract and to find a solution on how to configure the binding accordingly. This means that we need not only platform abstraction, but also platform awareness, which is the ability to characterise and exchange the properties of the platform. Systems which have such expression and negotiation capabilities with respect to non-functional properties are often termed Quality of Service Aware (c.f. [2]). However QoS research has mostly focused on static specification or dynamic negotiation tied to specific architectures.

This paper addresses how to expose varying platform capabilities in a way that facilitates negotiation between heterogeneous platforms on how to select suitable policies for bindings. The main contributions are: (1) A proposed model for declarative expressions. (2) A proposed scheme for how the middleware can support characterisation of resources or other relevant capabilities, as well as composition of these.

In section 2 we give an overview of the main ideas of our approach: Policy bindings negotiation and the need for a language for stating QoS requirements, and properties of the environment, and which supports conformance checking and composition. In section 3 we introduce our profile expression language. In section 4 we introduce our ideas for a negotiation support in our experimental middleware architecture. This includes dynamic profile expressions and a descriptor object framework. In section 5 we relate to current work in the area and in section 6 we conclude.

2. OVERVIEW

The main ideas of our approach, and the basis of our investigations are as follows:

- The concept of *policy* which define contract templates and contract enforcement plans. The concept of *metapolicy*, which define the management of bindings and associated policies [3].
- *Trading* of policy as a principle of negotiation and the use of declared conformance for matching property descriptions [4]. A language for *profile-expressions* used for exchanging requirements and environmental properties, and which support conformance checking and composition.
- Run-time expression support by the middleware.

2.1 Negotiation

We are interested in how to find a contract and a corresponding configuration when establishing a binding. We refer to such a process as negotiation since an overall goal is to reach agreement between possibly autonomous parties and since it may involve exchange of statements (offers and requirements).

Figure 1 illustrates which roles profile expressions play in negotiation based on policy trading. For a given service, application or application domain, there would exist a set of potential contracts, called policies, each stating an offer and an expectation. A contract is a promise, that the properties offered will be provided as long as the expectation is satisfied. The goal of negotiation is to find a policy whose offer (user profile) satisfies the user requirement while its expectation (service-profile) is satisfied by the environment properties (environment descriptor).

The relationships between user requirements and offers, and the relationships between the expectations and the capabilities of the environment, are satisfaction relationships. To facilitate conformance testing, a model should define a *partial order* on such expressions with respect to satisfaction. Then, any pair of expressions may be mechanically evaluated for conformance.

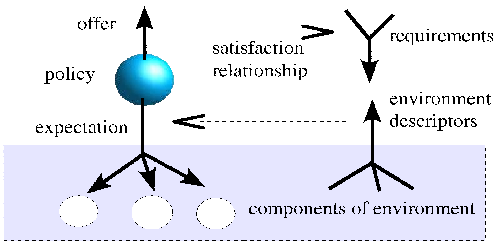


Figure 1. Statements and satisfaction relationships

2.1.1 Declared conformance

It looks appealing to adopt the technique typically used in ODP trading [5, 6] where each requirement or offer is a reference to a type name, and where a type conformance graph is declared a priori. This way of using declared conformance was first proposed in [7]. However, this is too limiting in general, since each declared type will need to capture all aspects relevant for the application. This may easily lead to conformance graphs which are too complex and application specific. Therefore we propose a hybrid model where conformance rules may also be based on simple numeric parameters.

2.1.2 Dynamic composition of statements

Profile expressions and negotiation should support *composition*, since statements from participants which do not necessarily know each other, would need to be combined into one describing the composed system. Given a set of expressions about the behaviour of individual components of a system, it is not obvious how to deduce the behaviour of the whole system. Three different problems should be addressed when it comes to expressing the total behaviour:

- Autonomous users may issue different requirements for the same service and all users should be satisfied.
- We may need to combine expressions regarding the same component but in more than one dimension (e.g. performance and security). We introduce an operator to construct expressions from simpler sub-expressions, meaning that the predicates stated by each part must be true in the same environment (c.f. logical conjunction).
- Open systems are systems interacting with environments neither they or their implementers controls [8]. Expectations towards environments may need to characterise a number of abstract components, for instance, client, server and communication channel, with a separate expectation for each of them. Our model should therefore support dynamic composition of statements about separate components of the environment. We address this problem by introducing an additional composing operator.

The third problem is only partly addressed in QoS specification models like [9], by allowing QoS-characteristics to be defined with composition in mind. Work on formal models has shown that with certain assumptions on the temporal relationships [10] it is possible to make statements about the behaviour of composite systems as conjunctions of statements about each component.

2.2 Binding model

The negotiation scheme discussed above need to be supported by a middleware architecture, it will be a part of the binding establishment process, where an active binding would represent a contract. The basis of our investigation is the family of binding models of ANSA [12], FlexiBind [3], OpenORB [13, 14], etc.

We believe that [14] is suitable as a generic binding model which regards binding-types as pluggable first-class entities. In our current experimental work, we assume a client/server (RMI) special case, and look at how client initiated binding would lead to a session specific end-to-end configuration. We also limit the scope of negotiation to the non-functional aspects. However we believe that the ideas explored here are applicable to other binding types as well.

2.2.1 Binding phases

We can decompose binding into four phases, where the system can perform configuration of a service implementation and where negotiation would be of interest:

- *Service deployment* (server side binding). A service is made available for clients to bind to, by generating a name and configuring a minimum of protocol stack such that client can establish bindings and initiate negotiations.
- *Client binding*, i.e. a client associates to the service. This would not necessarily lead to a complete configuration, since there may still be parts which need to be negotiated.
- *Activation*, where binding configuration as a result of negotiation is completed, and associated with necessary resources such that invocation may take place.
- Run-time *adaptation* by re-activation. Existing activations may be taken into account when re-negotiating the policy. It is also possible to encapsulate some adaptation within a single policy, if it does not violate the contract.

Note that we distinguish between component deployment (c.f. CCM, or EJB) and service deployment. Component deployment may involve service deployment.

2.2.2 Policy

A *policy* represents a potential *contract*, i.e. it can be viewed as a mapping from some constraint on the environment *S*, to the satisfaction of an user requirement U. We refer to *U* as the *user profile* and S as the *service profile*. If *P(x)* is the predicate defining a profile *x*, a policy states the following: $P(S) \Rightarrow P(U)$

A policy also constrains how an activation is configured. The configuration part of a policy will depend on the binding type. For RMI bindings it will consist of a client and a server part.

A *metapolicy* represent a way to associate policies with a given binding. A binding will always be associated with a metapolicy which constrain how and when it is activated, how the policy is negotiated, what scope a policy will have (e.g. invocation, session, transaction etc), and how the binding is adapted by re-activation in response to changing environmental properties.

Our concept of metapolicy capture how services are set up in the deployment phase as well as in the client binding phase. A metapolicy may therefore involve implementation decisions which constrain the later choice of policy.

3. PROFILE MODEL

In our approach statements about offers, expectations, etc. are formulated as *profile expressions* which can be evaluated for conformance. In this section we describe the idea of *basic profile models* and how more complex expressions can be composed from *basic profiles* by using *sum* or *component-sum* operators.

3.1 Defining basic profile models

A *basic profile* is an identifier and is associated with zero or more numeric parameters (parameters are enclosed in square brackets). A *profile model* define a set of rules for how basic profiles are related by conformance. If a profile x (implicitly) denotes a predicate $P(x)$, a conformance relationship exist: $x \leq y$, if $P(x) \Rightarrow P(y)$.

Since profile models only need to state conformance relationships, the actual meaning of a basic profile may be implicit in a profile model. Profiles can be abstractions over measurable properties like e.g. timing constraints, amounts of memory, but also structure of implementations etc. A policy programmer may however need a specification defining the actual meaning. For instance that `ModerateDelay` means average delay less than 500 milliseconds.

A concrete profile model is specified as a set of *axioms*. To define axioms we propose a simple notation like shown in the example below. Each axiom declares conformance between pairs of basic profiles using the '≤' operator. A predicate for when conformance is true is placed after the **'if'** keyword. Variables in the predicates are bound to the parameters given inside brackets. Omitting the predicate in a rule means **'true'** (corresponds to simple declared conformance).

```
NetGuaranteed ≤ NetEstimated ≤ Net;
LowLoad ≤ ModerateLoad ≤ HighLoad;
LowDelay ≤ ModerateDelay ≤ AnyDelay;
Delay[x] ≤ Delay[y], if x <= y;
Delay[x] ≤ LowDelay, if x <= 100;
XRes[x] ≤ XRes[y], if x <= y;
Disp[x1,y1] ≤ XRes[x], if x1 >= x;
Disp[x1,y1] ≤ Disp[x2,y2], if x1>=x2 AND y2>=y2;
```

From the rules above, we can for instance infer that the expression `Delay[10]` satisfy `ModerateDelay` and that `Disp [2000,1000]` satisfy `XRes[500]`.

As a proof of concept we have implemented a profile model compiler which checks the correctness of the definition, computes a set of additional rules which can be derived from the axioms. It generates code which facilitates efficient testing of conformance between any pair of basic profile expressions.

3.2 Composing expressions

3.2.1 Sum operator

Profile expressions can be combined using the '+' operator. The semantics of this operator is logical conjunction. If a profile expression x denotes (implicitly) a predicate $P(x)$, A profile expression $x+y$ denotes a predicate $P(x+y) = P(x) \wedge P(y)$.

From this definition it is straightforward to infer conformance. For instance $(x+y) \leq x$. Furthermore, $z \leq (x+y)$, if $z \leq x$ and $z \leq y$.

3.2.2 Component sum operator

The '⊕' (component sum) operator is used to state expressions regarding separate environments. To satisfy a component sum $x \oplus y$, both x and y must be satisfied, but x and y cannot be satisfied by the same profile instance. For a profile z to conform to $(x \oplus y)$, z must itself be a component sum $(a \oplus b)$ where $a \leq x$ and $b \leq y$.

A profile expression $(x \oplus y)$ denotes a predicate $P(x \oplus y) = P_1(x) \wedge P_2(y)$. where P is a composite of P_1 and P_2.

3.2.3 Expressions in general

From the definitions above we have developed a complete syntax and semantics of profile expressions formed by these operators. Based on this, we have developed conformance rules which can be used to match any expressions in this language. We refer to [11] for a more complete set of definitions and proofs.

A conformance testing algorithm has been implemented as a proof of concept. Conformance testing software (which will be part of policy trading software) will link in code generated by the profile model compiler.

3.3 Example

Consider an application for interactive browsing of graphics representing large and complex models (e.g. GIS). Clients initiate sessions to a server, and may have requirements for presentation quality and average response-time. Network connectivity and client device capabilities may vary, and the graphics rendering may put a high load on servers. The choice of policies for bindings will depend on user requirements and capabilities of client devices, servers and network.

A policy, which offers to satisfy low response time and a certain image quality may have the following expectation: A certain a minimum size of the display on the client side, a network connection satisfying an estimated *"NormalBW"* bandwith and latency better than 20, and a server with a "fast" CPU and a moderate load.

```
Client + ( (Display[800, 400] + Colour)
 ⊕ (NetEstimated + NormalBW + Delay[20] )
 ⊕ (Server + FastCPU + ModerateLoad) )
```

A client environment (e.g. a portable device) may for instance express that it is capable of two display modes: One normal colour mode and one monochrome mode with higher resolution, by including a component sum of two display instances:

```
(Display[200, 100] + Colour)
⊕ (Display[400, 800] + Mono)
```

4. MIDDLEWARE ARCHITECTURE

In this section we describe some highlights of our experimental middleware platform and how such a platform can support the run-time characterisation in the profile model described in section 3 above. The implementation is based on parts of FlexiBind [3].

4.1 Basic binding framework

Binders are pluggable components responsible for establishing bindings. Service deployment would mean associating objects with a suitable *generator* (server side binder), which generate an interface reference which can be *resolved* by a corresponding client side binder. Binders creates *bindings* which are not necessarily active. Bindings are associated with a metapolicy and are represented by explicit objects both on client and server side, or in all address spaces involved in the binding.

Activators are pluggable components responsible for activating bindings according to some policy. Activating a binding involves loading and instantiating an activator component. This may (depending on the policy) allocate resources needed by the activation, as well as setting up protocols, transparency objects, or other aspect implementation components. A policy will actually contain a reference to some activator implementation (a Java class in our experiment).

4.1.1 Channels

When a server generates an *interface reference* for an interface, some protocol information must be passed along with it such that clients know how to negotiate and bind to it. This observation suggest that the activation (protocol stack) should be divided into two parts: (1) A protocol dependent part which is identified in interface references, and a (2) protocol independent part which is negotiable. On the server side, the protocol dependent part should normally be the minimum needed to listen for incoming calls and to perform negotiation. On the client side, this means that we know what the protocol-dependent part of the activation should be at binding time, but it doesn't necessarily have to be activated before the rest of the stack is activated.

It is useful to encapsulate common protocol dependent configurations in components called *channels*. An instance of a middleware platform should offer access to one or more default channel instances and/or an interface to instantiate channels.

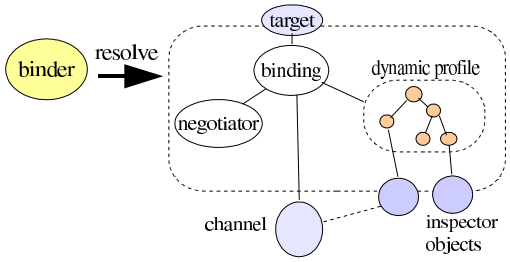
4.2 Negotiation aware bindings

Negotiation is handled by *negotiator* metaobjects which can be attached to bindings. They may intercept the methods for activation/deactivation to modify their behaviour. This essentially mean to add a mechanism for deciding on what activator to select. On the server side, binder would set up a negotiation metaobject which also export a special interface to be remotely invoked by clients to perform the negotiation. In our experiments this interface offer the following operations:

- *get_Activation*. Start the binding process on the server. It takes the user-requirement and the client side environment descriptor as arguments. It creates a prioritised list of candidate policies, and return the client part of the first policy which successfully is activated on the server.
- *retry_Activation*. Tell the server that the client part of the policy failed and that the server should try another one.
- *activation_OK*. Tell the server that the binding process has succeeded and that the server may now throw away the list of candidate policies.
- *release*. Close the binding.

A *policy-trading service* is located on the server, and it is used by the negotiation metaobject to compute a list of candidate policies. The trader is loaded with policies, each containing a reference to a client activator and a server activator. A corresponding client side binder would set up a negotiation metaobject (negotiator) which at first invocation or when explicitly requested, computes the two profile expressions to be sent with the *get_Activation* message. Figure 2 illustrates the binding setup where the *target* represent the actual application object on the server or a proxy object on the client.

The approach described here is one of several possible ways to design a negotiation protocol. It assumes a client/server model and that the probability of an activator failure is low. It can be extended to involve more components, for instance a three tier architecture with a backend server.



4.2.1 Interface references

The name of the protocol is part of interface references and is used to select a corresponding resolver component. In our initial scheme, the channel would identify the protocol. However, a server binder could also set up a negotiation scheme. Thererfore a protocol-id would be composed from two parts: One determined by the listening channel and one by the binder. Furthermore, the negotiation scheme described here will require two target identifiers, one for the actual target interface and one for the negotiation interface.

4.2.2 Dynamic profile expressions

We claim that it is a metapolicy issue how the environment-descriptors (c.f. section 2) are produced, since the relevance of properties would depend on the application, the binding type, the platform, the channel used, etc. As shown in figure 2, the binder would set up per binding instance, the necessary structures to produce such expressions.

Some parts of the descriptors may be static. This is the case for platform properties like display resolution or the availability of certain channels. However some properties may change due to varying load etc. Some may depend on the location of the peer, like for instance estimated end-to-end network delay. These cannot be fully provided before the time of negotiation. Therefore we propose a dynamic profile expression scheme: A binder will set up a profile expression tree (corresponds to an abstract parse tree). Parts of this may be dynamic, i.e. we use a special type of tree node which must be evaluated at negotiation time to get a complete expression. With this scheme we can easily set up the composition and the static parts as expressions embedded in the binder code.

4.2.3 Inspector objects

To support dynamic profiles we introduce *inspector* objects. Their role is to generate profile expression fragments describing platform specific facts or measurable properties of the system when requested. Inspectors offer an interface with a method *getProfile()* which returns an expression. A dynamic profile node would refer to an inspector, and inspectors may be shared between profile-expressions. Inspectors may be installed by platform configuration to report properties of platform wide resources, they

may be configured by channels, or they may be configured by binders to report properties of individual bindings.

Some of the inspectors would need to be configured with a *target object* (a reference to a local implementation or a remote interface reference). Other inspectors may not need to be associated with a target, but rather with the platform or resources available. Examples of what inspectors can do include:

- Estimate end-to-end invocation time by invoking probe-operations on the remote system. An inspector could e.g. return a profile "`RTT[n]`" (where *n* is a number denoting the round-trip time in milliseconds). More sophisticated implementations could use of policy specific interceptors or layers in the invocation chain which monitors the time for real operations, however requiring an existing activation.
- Determine by probing, if the remote system is reachable by the UDP protocol (not always the case if endsystems are on different IP-subnets). This can be useful if policies use UDP based invocation protocols or RTP for continous media streams.
- Estimate the load on the CPU, network interface or other resources on the platform. Such an inspector may make use of operating system specific services. For instance the CKRM module [15] for the Linux kernel provides class based reservation and monitoring of CPU, storage or listening sockets. A class could for instance guarantee that its members get a certain share of the resource. This can be used to determine in which class it is possible to place the threads of a session at a given time instance.

In the case of using class based resource management, actual reservations would be encapsulated in policies. The negotiation scheme cannot guarantee that reservation will succeed, unless the middleware is given exclusive access to the classes of interest by the O.S, and unless the negotiation protocol provides proper concurrency control wrt. resources of interest.

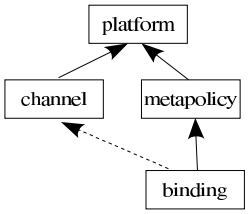
4.2.4 Naming and scoping support

Binder components are meant to be pluggable into various platform configurations. Hence, we want to abstract over how inspector objects are implemented and installed. We observe that (1) the platform might set up some, (2) channels might set up some, (3) binders set up some, and (4) some are metapolicy specific (set up by binders) but shared between the bindings sharing a metapolicy. Binders should be allowed to use and compose these objects.

This suggest that the middleware platform should support a naming and scoping mechanism for inspectors. Scoping is organised like in figure 3: The scope of a binding will also include the scope of the platform. We may want override a name defined in the platform scope. For instance, a metapolicy may wish to specialise the behaviour of a display inspector to reflect that only a part of the display could be used. It could then install a special inspector which delegates to the platform level display inspector but modifies its output.

4.2.5 Scripting

The use of a run-time naming and scoping mechanism leads us to the idea of defining dynamic profiles as script fragments embedded in binder code. It is convenient for a metapolicy programmer to embed textual representations of expressions and let the middleware evaluate it. In such expressions one could use the '$' prefix to refer to parts which are expanded at negotiation time. They would refer to installed inspectors by name.



4.2.6 Example

Recall the example in section 3.3. A client binder sets up an inspector named *'rpc-channel'* which returns the properties of an available RPC channel. An inspector named *'display'* is set up by the platform and returns the properties of the display. The client binder code would contain the following.

```
descriptor = "Client + ($display ⊕ $rpc-channel)"
```

During negotiation, this expression is evaluated, i.e. the dynamic profile parts are replaced by expressions returned by the inspectors, e.g. *$rpc-channel* estimates bandwidth and delay and return e.g. "`NetEstimated + HighBW + Delay[10]`". The resulting expression is sent to the server in the *get_Activation* operation and the server adds its own expression (evaluated in a similar way) by using the component sum operator. The resulting expression is then used when searching for a policy.

4.3 Implementation

The ideas presented here has been partially implemented. This includes binders, activators, a negotiator framework, an example negotiator pair, a simple policy-trader, dynamic profile-expression evaluation, an inspector framework and some example inspector and naming spaces which can be linked to provide proper scoping. Experiments using this implementation is currently being carried out.

We observe that the idea of naming and scoping have a wider application than only inspectors. In [3] we propose a extensible interface hierarchy for the PPI (policy programmer interface), which is used by policies to get access to services of the platform and which facilities the pluggability check of policies by using the dynamic type checking mechanism of the programming language. This scheme does not scale well wrt. number of possible platform configurations with different sets of services. It is not suitable for handling a varying number of instances of the same type, e.g. channels. This also indicates that one could benefit using declarative scripting, not only for composing dynamic profile expressions, but also for defining binders in general since different binders often represent sligthly different ways to configure and use a set of standard components.

5. RELATED WORK

Binding models in reflective middleware [1] is maturing. The ANSA FlexiNet framework [12] allows dynamic pluggability and selection of binders, [3] adds the concept of pluggable and replaceable policies for binding activation. The OpenORB binding model [14] focuses on extensibility wrt. binding types. Here, the client/server model is one of many specialisations. Since the concept of binding types includes a negotiation protocol, scope of binding etc, it overlap with our concept of metapolicy. The binding type will clearly constrain metapolicy, but it also seems like metapolicy would need to contain different aspects, some of them orthogonal to binding-type.

Much research has been done in QoS but is often tied to specific application domains, technologies, components or layered architectures (c.f. [16]). This includes QoS negotiation which is typically based on parameters and explicit constraints on parameter ranges, which may be computationally complex.

QuO [17] focuses on adaptation, contractual QoS and aspect languages. Contracts may be defined in a specific language, based on regions, constraining values on measured properties. Contracts are explicitly represented at run-time and closely tied to the server implementation. Furthermore this model does not address negotiation among autonomous components. QML [2] is mainly a language for QoS contract specification. A run-time representation is possible, however somewhat ad hoc. CQML [9] extends and generalises over this model and add some support for composition in the individual QoS characteristics. QML and CQML connect contract-templates to the service interfaces by use of so called profiles. We aim to make contracts more orthogonal to service types. Also, our approach offer a hybrid of declared and rule-based conformance instead of a strictly parameter based approach. Furthermore it addresses composition which is weakly supported in other approaches.

QuA [18] propose platform managed QoS as a general solution to preserve the safe deployment property for compositions of independently developed components. An important part of QuA is a framework for service planning [19], i.e. composing software components and resources to realise a service according to a set of QoS constraints. This is not far from the purpose of policy trading. QuA proposes to use a quality-loss model and utility functions, which has a more limited scope than our profile model but at the other hand, is suitable for maximising satisfaction in addition to just finding satisfactory contracts.

6. CONCLUDING REMARKS

We propose a model for declarative expressions to be used in negotiation of bindings in open systems. From application or domain specific rule-bases, we can infer conformance between pairs of expressions in this model. A compiler can derive a full set of rules and generate code which facilitates efficient conformance checking. Our model supports composition, i.e. conjoining of expressions describing separate components.

We also propose a scheme for how middleware can support automatic characterisation of resources or other relevant properties as well as composition of these. Each binding instance would be associated with a dynamic profile, i.e. a profile expression with placeholders for parts to be determined by querying at negotiation time. Such querying is done on inspector objects which perform mapping from platform dependent characteristics to the more abstract profile model. This means that QoS mapping is highly configurable and set up or modified by binder components. This scheme has the advantage of being flexible but requires some conventions for naming of inspectors. The profile model can simplify negotiation, and matching of policies can be more efficient than with more traditional parameter based negotiation, but requires careful design of profile models as well as conventions for composing expressions. A negotiation scheme strictly based on conformance does not support finding an optimal solution. That is a disadvantage in some cases.

Issues for future work in this area include validating this approach by applying it to application scenarios and alternative binding types. We observe that the metapolicy includes many aspects and that binders to a large extent share code. One could explore the use of declarative scripting languages for defining platform setup, binders, negotiators and activators. Since various applications or application domains may define their own profile models it is interesting to see how we can provide interoperability among autonomous domains by combining their models. Here, we may benefit from work performed in the area of semantic web with ontologies.

7. REFERENCES

[1] Kon, F., Costa, F., Blair, G and Campbell, R. H. *The Case for Reflective Middleware*. CACM June 2002/Vol. 46, No. 6.

[2] Frølund, S., and Koistinen, J. *Quality of Service Aware Distributed Object Systems,* Hewlett Packard Software Technology lab. report: HPL-98-142. 1998.

[3] Hanssen, Ø. and Eliassen, *F., A Framework for Policy Bindings*, In Proceedings of DOA'99, Edinburgh, IEEE press,

[4] Hanssen, Ø. and Eliassen, F., *Policy Trading*, In Proceedings of DOA'00, Antwerp, IEEE press, 2000.

[5] Bearman, M. Y., *ODP Trader*, In Proceedings of ICODP'93, Berlin, 1993

[6] *ODP Trading Function*, Report, ITU-T X.950 – ISO/IEC 13235.

[7] Hanssen, Ø. and Eliassen, F., *Towards a QoS aware Binding Model*, In Proceedings of SYBEN'98, Zurich, Spie press, 1998.

[8] Abadi, M., and Lamport, L. Open Systems in TLA, In Proceedings of ACM Symposium on Principles of Distributed Computing, August 1994.

[9] Aagedal, J. Ø., *Quality of Service Support in development of Distributed Systems,* Ph.D. Thesis, University of Oslo, 2001.

[10] Abadi, M., and Lamport, L. *Conjoining Specifications*, Digital Systems Research Center, Report 118.

[11] Hanssen, Ø. *A Declarative Profile Model for QoS Negotiation*. Technical report 2005-54, University of Tromsø, Computer Science Department, 2005.

[12] Hayton, R. and Herbert, A., *FlexiNet: A Flexible, Component-Oriented Middleware System*, Lecture notes in Computer Science, 1752, p. 497 ff, Springer Verlag, 2000.

[13] Blair, G.S., et al. The Design and implementation of Open ORB 2, IEEE Distributed Systems Online, 2, (2001), no. 6.

[14] Parlavantzas, N., Coulson, G. and Blair, G.S., *An extensible Binding Framework for Component-Based Middleware*, In Proceedings of EDOC 2003.

[15] Nagar, S., et al., Improving Linux resource control using CKRM, In Proceedings of the Linux Symposium, Vol two, July, 2004.

[16] Ecklund, D., et al., *QoS Management Middleware: A Separable, Reusable Solution*, In Proceedings of IDMS 2001, LNCS 2158, pp. 124-137, Springer Verlag 2001.

[17] Loyall, D.E.,et al. *Specifying and Measuring Quality of Service in Distributed Object Systems*, In Proc. ISORC'98, IEEE press 1998.

[18] Staehli, R. and Eliassen, F., *A QoS Aware Component Architecture*, Simula Research Laboratory Research report, 2002 – 12.

[19] Solberg, A., Amundsen, S., Aagedal, J.Ø. and Eliassen, F., *A Framework, for QoS-Aware Service Composition*, In Proceedings of 2nd ACM Intl. Conference on Service Oriented computing, ICSOC 2004.